

BRIEF CONTENTS

Acknowledgments xix

Introduction xxi

PART I: THE TECHNOLOGY STACK.1

Chapter 1: Node.js 3

Chapter 2: Modern JavaScript. 15

Chapter 3: TypeScript. 33

Chapter 4: React 53

Chapter 5: Next.js 69

Chapter 6: REST and GraphQL APIs. 93

Chapter 7: MongoDB and Mongoose. 115

Chapter 8: Testing with the Jest Framework 129

Chapter 9: Authorization with OAuth 157

Chapter 10: Containerization with Docker 173

PART II: THE FULL-STACK APPLICATION183

Chapter 11: Setting Up the Docker Environment 185

Chapter 12: Building the Middleware. 195

Chapter 13: Building the GraphQL API. 207

Chapter 14: Building the Frontend 215

Chapter 15: Adding OAuth. 231

Chapter 16: Running Automated Tests in Docker 253

Appendix A: TypeScript Compiler Options 259

Appendix B: The Next.js app Directory. 263

Appendix C: Common Matchers 289

Index 295

CONTENTS IN DETAIL

ACKNOWLEDGMENTS

xix

INTRODUCTION

xxi

Who Should Read This Book?	xxii
What's in This Book?	xxii
The Parts of a Full-Stack Application	xxiv
The Frontend	xxiv
The Middleware	xxv
The Backend	xxv
A Brief History of JavaScript and Full-Stack Development.	xxvi
Setting Up.	xxvi

PART I: THE TECHNOLOGY STACK

1

1

NODE.JS

3

Installing Node.js	4
Working with npm	4
The package.json File	4
Required Fields	5
Dependencies	6
Development Dependencies	6
The package-lock.json File	6
Creating a Project	8
Initializing a New Module or Project	8
Installing the Dependencies	8
Installing the Development Dependencies	9
Auditing the package.json File	10
Cleaning Up the node_modules Folder	11
Updating All Packages	11
Removing a Dependency	11
Installing a Dependency	11
Using npx to Execute a Script Only Once	12
Exercise 1: Build a "Hello World" Express.js Server	13
Setting Up	13
Writing the Server Code	13
Summary	14

2

MODERN JAVASCRIPT

15

ES.Next Modules	15
Using Named and Default Exports	16
Importing Modules	17

Declaring Variables	17
Hoisted Variables	18
Scope-Abiding Variables	19
Constant-Like Data.	20
Arrow Functions.	20
Writing Arrow Functions	21
Understanding Lexical Scope	21
Exploring Practical Use Cases.	22
Creating Strings.	22
Asynchronous Scripts	24
Avoiding Traditional Callbacks	24
Using Promises	25
Simplifying Asynchronous Scripts	26
Looping Through an Array	27
Dispersing Arrays and Objects	27
Exercise 2: Extend Express.js with Modern JavaScript	29
Editing the package.json File	29
Writing an ES.Next Module with Asynchronous Code	29
Adding the Modules to the Server.	30
Summary	31

3

TYPESCRIPT	33
Benefits of TypeScript	34
Setting Up TypeScript	36
Installation in Node.js	36
The tsconfig.json File	37
Dynamic Feedback with TypeScript	38
Type Annotations	38
Declaring a Variable	39
Declaring a Return Value	39
Declaring a Function's Parameters.	39
Built-in Types	40
Primitive JavaScript Types.	40
The union Type	41
The array Type	41
The object Type.	42
The tuple Type	42
The any Type	43
The void Type.	43
Custom Types and Interfaces	44
Defining Custom Types.	44
Defining Interfaces	45
Using Type Declaration Files.	45
Exercise 3: Extend Express.js with TypeScript.	46
Setting Up	46
Creating the tsconfig.json File.	47
Defining Custom Types.	47
Adding Type Annotations to the routes.ts File	48
Adding Type Annotations to the index.ts File.	49
Transpiling and Running the Code.	50
Summary	51

4	REACT	53
The Role of React		53
Setting Up React		55
The JavaScript Syntax Extension		56
An Example JSX Expression		56
The ReactDOM Package		57
Organizing Code into Components		57
Writing Class Components		59
Providing Reusable Behavior with Hooks		61
Working with Built-in Hooks		62
Managing the Internal State with useState		62
Handling Side Effects with useEffect		62
Sharing Global Data with useContext and Context Providers		63
Exercise 4: Create a Reactive User Interface for the Express.js Server		64
Adding React to the Server		64
Creating the Endpoint for the Static HTML File		66
Running the Server		66
Summary		67
5	NEXT.JS	69
Setting Up Next.js		70
Project Structure		71
Development Scripts		72
Routing the Application		72
Simple Page Routes		73
Nested Page Routes		73
API Routes		75
Dynamic URLs		77
Styling the Application		78
Global Styles		79
Component Styles		79
Built-in Next.js Components		80
The next/head Component		80
The next/link Component		81
The next/image Component		82
Pre-rendering and Publishing		83
Server-Side Rendering		84
Static Site Generation		86
Incremental Static Regeneration		87
Client-Side Rendering		88
Static HTML Exporting		89
Exercise 5: Refactor Express.js and React to Next.js		89
Storing Custom Interfaces and Types		90
Creating the API Routes		90
Creating the Page Routes		90
Running the Application		91
Summary		91

6		
REST AND GRAPHQL APIS		93
REST APIs		94
The URL		95
The Specification		95
State and Authentication		97
HTTP Methods		98
Working with REST		99
Reading Data		99
Updating Data		100
GraphQL APIs		101
The Schema		101
The Resolvers		103
Comparing GraphQL to REST		106
Over-Fetching		106
Under-Fetching		107
Exercise 6: Add a GraphQL API to Next.js		108
Creating the Schema		108
Adding Data		109
Implementing Resolvers		109
Creating the API Route		110
Using the Apollo Sandbox		111
Summary		113
7		
MONGODB AND MONGOOSE		115
How Apps Use Databases and Object-Relational Mappers		116
Relational and Non-Relational Databases		116
Setting Up MongoDB and Mongoose		117
Defining a Mongoose Model		118
The Interface		118
The Schema		118
The Model		119
The Database-Connection Middleware		120
Querying the Database		121
Creating a Document		121
Reading a Document		122
Updating a Document		122
Deleting a Document		123
Creating an End-to-End Query		123
Exercise 7: Connect the GraphQL API to the Database		125
Connecting to the Database		125
Adding Services to GraphQL Resolvers		126
Summary		127
8		
TESTING WITH THE JEST FRAMEWORK		129
Test-Driven Development and Unit Testing		130
Using Jest		130
Creating an Example Module to Test		131

Anatomy of a Test Case	132
Arrange	132
Act	133
Assert	133
Using TDD.	135
Refactoring Code	136
Evaluating Test Coverage	138
Replacing Dependencies with Fakes, Stubs, and Mocks	139
Creating a Module with Dependencies	140
Creating a Doubles Folder	141
Using a Stub.	142
Using a Fake	142
Using a Mock.	143
Additional Types of Tests	144
Functional Tests	144
Integration Tests	144
End-to-End Tests.	145
Snapshot Tests	145
Exercise 8: Add Test Cases to the Weather App	146
Testing the Middleware with Spies	146
Creating Mocks to Test the Services	148
Performing an End-to-End Test of the REST API.	151
Evaluating the User Interface with a Snapshot Test.	153
Summary	156

9 AUTHORIZATION WITH OAUTH 157

How OAuth Works.	158
Authentication vs. Authorization	158
The Role of OAuth.	159
Grant Types	159
Bearer Tokens.	160
The Authorization Code Flow	161
Creating a JWT Token	163
The Header	163
The Payload	163
The Signature	166
Exercise 9: Access a Protected Resource	168
Setting Up the Client	168
Logging In to Receive the Authorization Grant	170
Using the Authorization Grant to Get the Access Token	171
Using the Access Token to Get the Protected Resource	172
Summary	172

10 CONTAINERIZATION WITH DOCKER 173

The Containerization Architecture	174
Installing Docker	174

Creating a Docker Container	174
Writing the Dockerfile	175
Building the Docker Image	176
Serving the Application from the Docker Container	177
Locating the Exposed Docker Port	177
Interacting with the Container	178
Creating Microservices with Docker Compose	178
Writing the docker-compose.yml File	179
Running the Containers	180
Rerunning the Tests	181
Interacting with Docker Compose	182
Summary	182

PART II: THE FULL-STACK APPLICATION 183

11 SETTING UP THE DOCKER ENVIRONMENT 185

The Food Finder Application	186
Building the Local Environment with Docker	186
The Backend Container	186
The Frontend Container	189
Summary	193

12 BUILDING THE MIDDLEWARE 195

Configuring Next.js to Use Absolute Imports	196
Connecting Mongoose	196
Writing the Database Connection	197
Fixing the TypeScript Warning	198
The Mongoose Model	199
Creating the Schema	199
Creating the Location Model	201
The Model's Services	202
Creating the Location Service's Custom Types	203
Creating the Location Services	203
Testing the Services	206
Summary	206

13 BUILDING THE GRAPHQL API 207

Setting Up	208
The Schemas	208
The Custom Types and Directives	208
The Query Schema	209
The Mutation Schema	209
Merging the Typedefs into the Final Schema	209
The GraphQL Resolvers	210
Adding the API Endpoint to Next.js	212
Summary	214

14	BUILDING THE FRONTEND	215
Overview of the User Interface		215
The Start Page		216
The List Item		216
The Locations List		218
The Page		219
The Global Layout Components		222
The Logo		222
The Header		223
The Layout		224
The Location Details Page		227
The Component		227
The Page		228
Summary		230
15	ADDING OAUTH	231
Adding OAuth with next-auth		231
Creating a GitHub OAuth App		232
Adding the Client Credentials		232
Installing next-auth		233
Creating the Authentication Callback		233
Sharing the Session Across Pages and Components		235
The Generic Button Component		235
The AuthElement Component		238
Adding the AuthElement Component to the Header		241
The Wish List Next.js Page		243
Adding the Button to the Location Detail Component		244
Securing the GraphQL Mutations		247
Summary		252
16	RUNNING AUTOMATED TESTS IN DOCKER	253
Adding Jest to the Project		254
Setting Up Docker		254
Writing Snapshot Tests for the Header Element		256
Summary		257
A	TYPESCRIPT COMPILER OPTIONS	259
B	THE NEXT.JS APP DIRECTORY	263
Server Components vs. Client Components		264
Server Components		264
Client Components		265

Rendering Components	266
Fetching Data	266
Static Rendering	267
Dynamic Rendering	268
Exploring the Project Structure	269
Updating the CSS	271
Defining a Layout	273
Adding the Content and Route	275
Catching Errors	277
Showing an Optional Loading Interface	279
Adding a Server Component That Fetches Remote Data	281
Completing the Application with the Navigation	284
Replacing API Routes with Route Handlers	285

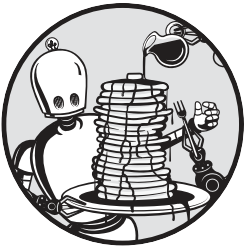
C

COMMON MATCHERS 289

Built-in Matchers	289
The JEST-DOM Matchers	292

INDEX 295

INTRODUCTION



Nearly all programming jobs today require at least a cursory understanding of full-stack development, but if you're a beginner, you might struggle to find the right entry point to this overwhelming topic. You might not even know what the term means.

Simply put, *full-stack web development* typically refers to the creation of complete web applications using JavaScript and the many frameworks built for it. It requires a mastery of the traditional disciplines of frontend and backend development, as well as the ability to write middleware and various kinds of application programming interfaces (APIs).

Lastly, a well-rounded full-stack developer can handle databases and has professional skills, such as the ability to craft automated tests and deploy their code by themselves. To do all of this, they must understand HTML, CSS, and JavaScript, as well as the language's typed counterpart, TypeScript. For a crash course on some of this terminology, see “The Parts of a Full-Stack Application” on page xxiv.

If this sounds like a lot, you’ve come to the right place. This book will introduce you to each component of a modern application and teach you how to use some of the most widely used technologies to build them.

Who Should Read This Book?

There are two primary audiences for the book. The first includes professional frontend or backend engineers who want to advance their careers by mastering full-stack development. The second includes inexperienced, beginning developers interested in learning about web development.

While the book introduces many technologies from scratch, it assumes some prior familiarity with HTML, CSS, and JavaScript, as well as the client/server architecture of most web applications. For a refresher, see *The Coding Workbook* by Sam Taylor (No Starch Press, 2020), which teaches you how to build a website with HTML and CSS, and *The Book of CSS3*, 2nd edition, by Peter Gasston (No Starch Press, 2014) to sharpen your CSS skills. To familiarize yourself with JavaScript, I recommend *JavaScript Crash Course* by Nick Morgan (No Starch Press, 2024), which is a fast-paced JavaScript tutorial for beginners, and *Eloquent JavaScript*, 3rd edition, by Marijn Haverbeke (No Starch Press, 2018), for a deep dive into JavaScript.

What’s in This Book?

The book is split into two parts. Part I, comprising Chapters 1 through 10, introduces you to the components of a modern technology stack. Each chapter focuses on one technology and highlights the topics you need to know as a full-stack developer. The exercises will encourage you to begin writing application code from page 1.

Chapter 1: Node.js Introduces you to Node.js and its ecosystem, which let you run JavaScript code outside a browser. Then you’ll use Node.js and the Express.js framework to create your own simple web server with JavaScript.

Chapter 2: Modern JavaScript Focuses on contemporary JavaScript syntax useful for full-stack developers, including how to use modules to write maintainable code packages. We look at the different ways to define variables and constants, the arrow function, and techniques for asynchronous code. You’ll use these to rewrite your JavaScript server.

Chapter 3: TypeScript Introduces TypeScript, a superset of JavaScript, and highlights how modern full-stack development benefits from it. We discuss the shortcomings and pitfalls of JavaScript and how to effectively leverage TypeScript’s type system through inference. You’ll conclude the chapter by refactoring your JavaScript server with type annotations, custom types, and interfaces.

Chapter 4: React Discusses React, one of the most common libraries for creating user interface components. You’ll see how its components simplify full-stack development and learn how to use its JSX elements,

the virtual DOM, and hooks. You'll then use React to add a reactive user interface to your Express.js server.

Chapter 5: Next.js Focuses on Next.js, the leading web application framework built on top of React. You'll create pages and custom API routes with Next.js's file-based routing before learning different ways to render a page within the framework. Finally, you'll migrate the Express.js server to Next.js as an exercise.

Chapter 6: REST and GraphQL APIs Teaches you all about APIs, what they are, and how to use them for full-stack web development. We explore two kinds of APIs: REST and GraphQL. You'll conclude the chapter by adding an Apollo GraphQL server to your Next.js full-stack application.

Chapter 7: MongoDB and Mongoose Discusses the differences between traditional relational databases and non-relational databases such as MongoDB. You'll add the Mongoose object data modeling tool to your technology stack to simplify working with a database. You'll then connect the GraphQL API to your own MongoDB database.

Chapter 8: Testing with the Jest Framework Explains the importance of automated tests and test-driven development to full-stack development. We explore different types of tests, common test patterns, and the concepts of test doubles, stubs, fakes, and mocks. Lastly, you'll add a few basic snapshot tests to your Next.js application with the Jest framework.

Chapter 9: Authorization with OAuth Discusses authentication and authorization and how full-stack developers can use the OAuth protocol to handle those tasks by integrating with a third-party service. We walk through this authorization flow and its components. You'll run through a complete OAuth interaction on the command line to explore each step in depth.

Chapter 10: Containerization with Docker Introduces you to using Docker to deploy your application. We cover the concept of a microservice architecture, then cover all relevant components of the Docker ecosystem: the host, the Docker daemon, Dockerfiles, images, containers, volumes, and Docker Compose. You'll conclude by splitting your application into self-contained microservices.

In Part II, you'll use your newfound knowledge to build a web application that applies the concepts, tools, and frameworks introduced in Part I. The Food Finder application is a location search service that lets users log in with their GitHub account and maintain a wish list of places to visit.

Chapter 11: Setting Up the Docker Environment Create the foundation of your Food Finder application by using your knowledge of Docker and containerization to set up your development environment. You'll use Docker Compose to decouple the application development from your local system and then add a MongoDB server as its own service.

Chapter 12: Building the Middleware Create the first part of the Food Finder application's middleware. Here you'll connect Mongoose to the MongoDB service and create its schema, model, services, and

custom types. With these pieces in place, you'll be able to create, read, update, and delete data from your database.

Chapter 13: Building the GraphQL API Use your knowledge of GraphQL to add an Apollo GraphQL server to your Food Finder application, then implement a public GraphQL API. You'll be able to use the Apollo sandbox to read and update data with GraphQL on your MongoDB server.

Chapter 14: Building the Frontend Use React components and the Next.js framework to build the frontend for the Food Finder application. At this point, you'll have implemented a complete modern full-stack application that reads data from the database through your custom middleware and renders the data to your application's frontend.

Chapter 15: Adding OAuth Add an OAuth flow to your app so that visitors can log in to maintain a personal wish list of locations. You'll use the *next-auth* package from Auth.js to add login options using GitHub.

Chapter 16: Running Automated Tests in Docker Set up automated snapshot tests with Jest and configure a new service to run the tests automatically.

Then, in the appendices, you'll get detailed information on the TypeScript Compiler options and the most common Jest matchers. Also, you'll use your newfound knowledge to explore and understand Next.js's modern app directory approach.

Appendix A: TypeScript Compiler Options Shows the most common TypeScript Compiler (TSC) options so that you can customize your own TypeScript projects to your liking.

Appendix B: The Next.js app Directory Explores a new routing pattern using the *app* directory that Next.js introduced in version 13. You can then choose to work with either the traditional pages approach covered in Chapter 5 or the modern *app* directory in your own upcoming projects.

Appendix C: Common Matchers Shows the most common matchers for testing your applications with Jest and the Jest DOM.

The Parts of a Full-Stack Application

Throughout this book, we'll discuss various portions of an application. This section gives you a crash course on what we mean when we use the terms *frontend*, *middleware*, and *backend*.

The Frontend

The frontend is the user-facing part of a website or web application. It runs on the client, typically a web browser. You can think of it as the “front

office” of the web application. For example, on <https://www.google.com>, the frontend is a page with a simple search bar, though of course, frontend development can be much more complex than this; take a look at Google’s search results page or the interface of the last website you visited.

Frontend developers focus on user engagement, experiences, and interfaces. They rely on HTML for creating the elements of the website’s interface, CSS for styling, JavaScript for user interactions, and frameworks such as Next.js to pull everything together.

The Middleware

The middleware connects an application’s frontend and backend and performs all of its chores, such as integrating with third-party services and transferring and updating data. You can think of it as the employees on the company floor.

As full-stack developers, we often write middleware for *routing* our applications, which means serving the correct data for a particular URL, handling database connections, and performing authorization. For example, on <https://www.google.com>, the middleware asks the server for the landing page’s HTML. Then a different part of the middleware checks whether the user is logged in, and if so, which personal data it should show. Meanwhile, a third part of the middleware consolidates the information from each of these data streams and then answers the server’s requests with the correct HTML.

One essential part of a full-stack application’s middleware is its *API layer*, which exposes the application’s APIs. Generally, an API is code written to connect two machines. Often, an API lets the frontend code (or a third party) access the application’s backend. JavaScript-driven development relies on two primary architectural frameworks for creating APIs: REST and GraphQL, both of which are covered in Chapter 6.

You could write the middleware by using any programming language. Most full-stack developers use modern JavaScript or TypeScript, but they could instead use PHP, Ruby, or Go.

The Backend

The backend is the invisible part of a web application. In a JavaScript-driven application, the backend runs on a server, typically Express.js, though others might use Apache or NGINX. You can think of it as the “back office” of the web application.

More concretely, the backend handles any operations involving the application’s data. It performs create, read, update, and delete (CRUD) operations on the values stored in the database and returns the datasets requested by the user through the middleware’s API layer. For <https://www.google.com>, the backend is the code that searches the database for the keywords you entered in the frontend, which the backend received through the middleware. The middleware would combine these search results with other relevant pieces of information. Then the user would see the search results page rendered by the frontend.

Backend development can be done in any programming language. Full-stack developers usually employ modern JavaScript or TypeScript. Other options include PHP, Ruby, Elixir, Python, Java, and frameworks like Symfony, Ruby on Rails, Phoenix, and Django.

A Brief History of JavaScript and Full-Stack Development

All developers should understand the context of the tools they're using. Before we begin developing, let's start with a bit of history.

The full-stack developer position evolved alongside JavaScript, which began as nothing more than a scripting language that ran in users' browsers. Developers used it to add elements to their websites, such as accordions, pop-up menus, and overlays, that reacted immediately to a user's behavior, without requiring requests to the application's server.

Until the late 2000s, most JavaScript libraries were designed to provide consistent interfaces to handle vendor-specific quirks. Often, the JavaScript engines were slow, especially when interacting with, updating, or modifying the HTML. Hence, JavaScript was considered a quirky scripting language for the frontend and was frowned upon by backend developers.

Several projects attempted to popularize the use of JavaScript in the backend, but until the release of Node.js in 2009, these didn't gain any traction. Node.js, covered in Chapter 1, is a JavaScript tool for developing backends. Shortly thereafter, the Node.js package manager npm built the missing ecosystem for full-stack JavaScript development.

This ecosystem includes a host of JavaScript libraries for working with databases, building user interfaces, and writing server-side code (many of which we'll explore in this book). These new tools allowed developers to use JavaScript reliably on both the client and the server. Of particular importance, Google released the Angular framework in 2010, and Meta (known as Facebook at the time) released React in 2013. The commitment of these internet giants to building JavaScript tools turned full-stack web development into a sought-after role.

Setting Up

Throughout this book, you'll write code and run command line tools. You can use any development environment you'd like, but here are some guidelines.

The most common code editor these days is Visual Studio Code, which you can download from <https://code.visualstudio.com>. It is Microsoft's open source editor and is free for Windows, macOS, and Linux. In addition, you can extend and configure it through a plethora of third-party plug-ins and adjust its appearance to your liking. However, if you're used to a different editor, such as Vim or Emacs, you can keep using it. The book doesn't require a particular tool.

Depending on your operating system, your default command line program will be either the *Command Prompt* (on Windows) or the *Terminal* (on

macOS and Linux). These programs use slightly different syntax for tasks like creating, changing, and listing the contents of a directory. This book shows the Linux and macOS versions of these commands. If you're using Windows, you'll have to adapt the commands for your operating system. For example, instead of `ls`, Windows uses `dir` to list files and folders in the current directory. Microsoft's official command line reference lists all available commands here: <https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands#command-line-reference-a-z>.

The most notable difference between operating systems relevant to this book is the escape character used for line breaks in multiline cURL commands. This escape character is `\` on macOS and `^` on Windows. We'll point out these differences in Chapter 6, when we first use cURL.

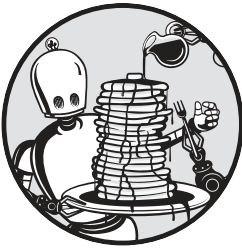
You can download the code listings for the first part of the book and the complete source code for the Food Finder application from <https://www.usemodernfullstack.dev/downloads>.

PART I

THE TECHNOLOGY STACK

1

NODE.JS



Node.js is an open source runtime environment that executes JavaScript code outside a web browser. You could, for example, use it as a scripting language to perform all kinds of chores, such as deleting and moving files, logging data on the server side, or even creating your own web server (as we'll do in this chapter's exercise).

Knowing how to use Node.js is not really about understanding individual commands or packages, because it relies on standard JavaScript and you can refer to the documentation for details about its syntax and parameters. Instead, all developers should strive to understand the Node.js ecosystem and use it to their advantage. This chapter will introduce you to it.

Installing Node.js

Begin by checking whether Node.js is already available on your local machine by running the `node` command from your command line. The version flag (`-v`) should return the current Node.js version:

```
$ node -v
```

If you see an output with a version number, Node.js is installed. If you don't, or if the version is lower than the currently recommended stable release listed on <https://nodejs.org>, you should install this stable version.

To install Node.js locally, go to <https://nodejs.org/en/download> and select the installer for your operating system. I recommend installing the long-term support (LTS) version of Node.js because many Node.js modules require this version. Run the installer package for Node.js LTS and npm, then check the version number again. It should match the one you've just installed.

Next, we'll review the basic commands and features of the Node.js runtime environment. If you prefer not to install Node.js, you can run the Node.js command line examples and JavaScript code in the online playgrounds at <https://codesandbox.io/s/new> and <https://stackblitz.com>.

Working with npm

The default package manager for Node.js is npm. You can find modules for every task there, taken from the online registry at <https://www.npmjs.com>. Verify that npm is available on your local machine by running the following on the command line:

```
$ npm -v
```

If there is no listed version or if the version is lower than the current release, install the latest Node.js LTS version, including npm.

Be aware that there is no vetting process or quality control on <https://www.npmjs.com>. Anyone can publish packages, and the site relies on the community to report any that are malicious or broken.

Running the following shows a list of available commands:

```
$ npm
```

NOTE

The most popular alternative to npm is yarn, which also uses the <https://www.npmjs.com> registry and is fully compatible with npm.

The package.json File

The `package.json` file is a key element of each Node.js-based project. While the `node_modules` folder contains actual code, the `package.json` file holds all

the metadata about the project. Found in the project's root, it must contain the project's name and version; in addition, it can contain optional data, such as the project's description, a license, scripts, and many more details.

Let's take a look at the *package.json* file for the web server you'll create in Exercise 1 on page 13. It should look similar to the one shown in Listing 1-1.

```
{
  "name": "sample-express",
  "version": "1.0.0",
  "description": "sample express server",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "run": "node index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Listing 1-1: The package.json file for the Express.js server project in Exercise 1

The *package.json* file includes all the information others will need to install required modules on their machine and run the application. As a result, you never have to ship or store the *node_modules* folder in your code repository, which minimizes the repository's size. Let's take a detailed look at the *package.json* file.

Required Fields

The *package.json* file must contain a name field and a version field. All other fields are optional. The name field contains the package's name, which must be one lowercase word but can contain hyphens and underscores.

The version field must follow semantic versioning guidelines, which suggest this format: *major.minor.patch*; for example, *1.2.3*. We call this *semantic* versioning because each number conveys a meaning. A *major* version introduces an incompatible API change. You should generally be very careful about switching to another major version, as you won't be able to expect that your application will work flawlessly. A *minor* version change adds new functionality in a backward-compatible manner and therefore shouldn't pose problems for your application. A *patch* version applies backward-compatible bug fixes, and you should always keep it up to date.

NOTE

You can read more about semantic versioning and how to define different ranges at <https://semver.org>.

Dependencies

The most important optional fields specify the dependencies and development dependencies. The `dependencies` field lists all the dependencies needed to run the project, along with their required version ranges, following the semantic versioning syntax. By default, npm requires only the major version and keeps the minor and patch ranges flexible. This way, npm can always initialize your project with the latest compatible version.

These dependencies are part of your bundled application. When you install a project on a new machine, all dependencies listed in the `package.json` file will be installed and placed in the `node_modules` folder, next to `package.json`.

Your application could require all sorts of dependencies, such as frameworks and helper modules. For example, the Food Finder application we'll build in Part II must contain at least Next.js as a single-page application framework, and Mongoose with MongoDB for the database layer.

Development Dependencies

The `devDependencies` field lists all the dependencies necessary to develop the project, along with their versions. Again, only the major version is fixed. These are required only to develop, and not to run, the application. Hence, they are ignored by the packaging scripts and are not part of the deployed application. When you install a project on a new machine, all the development dependencies listed in the `package.json` file will be installed and placed in the `node_modules` folder next to `package.json`. For our Food Finder application, our development dependencies will include TypeScript's type definitions. Other typical entries are testing frameworks, linters, and build tools such as webpack and Babel.

The package-lock.json File

The npm package manager automatically generates the `package-lock.json` file for each project. This lock file resolves a problem introduced by the use of semantic versioning for dependencies. As mentioned earlier, the npm default is to define only the major version and to use the latest minor and patch versions available. While this ensures that your application includes the latest bug fixes, it introduces a new issue: without an exact version, builds aren't reproducible. Because there's no quality control in the npm registry, even a patch or minor version update could introduce an incompatible API change that should have been a major version change. Consequently, a slight deviation between versions could result in a broken build.

The `package-lock.json` file solves this by tracking the exact version of every package and its dependencies. This file is usually quite big, but its entries for the web server you'll create at the end of this chapter will look similar to Listing 1-2.

```

{
  "name": "sample-express",
  "lockfileVersion": 2,
  "requires": true,
  "packages": {
    "": {
      "dependencies": {
        "express": "^4.18.2"
      }
    },
    "node_modules/accepts": {
      "version": "1.3.8",
      "resolved": "https://registry.npmjs.org/accepts/-/accepts-1.3.8.tgz",
      "integrity": "sha512-PsL6iGPdFq/LKM1Uuie1Ygv3BUoJfz1aUwU9vHZ+J7gyvwdQXFEBIEI==",
      --snip--
    },
    --snip--
    "node_modules/express": {
      "version": "4.18.2",
      "resolved": "https://registry.npmjs.org/express/-/express-4.18.2.tgz",
      "integrity": "sha512-Psl6iGPdFq/LKM1Uuie1Ygv3BUoJfz1aUwU9vHZ+J7gyvwdQXFEBIEI==",
      "dependencies": {
        "accepts": "~1.3.8",
        --snip--
        "vary": "~1.1.2"
      },
      "engines": {
        "node": ">= 0.10.0"
      }
    },
    --snip--
    "vary": {
      "version": "1.1.2",
      "resolved": "https://registry.npmjs.org/vary/-/vary-1.1.2.tgz",
      "integrity": "sha512-BNGbWLfdoeUPabhkXUVmOj8uuvREyTh5ovRa/dyow/BqAbZJyC+bfhskkh=="
    }
  }
}

```

Listing 1-2: The package-lock.json file for Exercise 1

The lock file contains a reference to the project and lists the information from the corresponding *package.json* file. Then it lists all the project's dependencies; for us, the only dependency is Express.js, with a pinned version. (We'll cover Express.js in Exercise 1.) In addition, the file lists all the dependencies for the Express.js version in use, in this case the *accept* and *vary* packages. The stored artifact's SHA hash enables npm to verify the integrity of the resource after downloading it.

Now, with all modules version-locked, every `npm install` command will create an exact clone of the original setup. Like *package.json*, the *package-lock.json* file is part of the code repository.

Creating a Project

Let's cover the most important commands for your day-to-day work, in the order in which you would logically use them to create and maintain a project. After performing these steps, you'll have a *package.json* file and a production-ready project folder with one installed package, Express.js.

Initializing a New Module or Project

To start a new project, run `npm init`, which initializes a new module. This should trigger an interactive guide through which you'll populate the project's *package.json* file based on your input:

```
$ mkdir sample-express
$ cd sample-express
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.
--snip--
Is this OK? (yes)
```

At the beginning of each project, you need to initialize a new Node.js setup in an empty folder (created here with `mkdir sample-express`) using `npm init`. For simplicity, keep the default suggestions here. The assistant creates a basic *package.json* file in your project folder. It should look similar to Listing 1-3.

```
{
  "name": " sample-express",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Listing 1-3: The default package.json file

When we compare this file with the one shown in Listing 1-1, we see that they are fairly similar, except for the dependencies and development dependencies. With the *package.json* file ready, we can now install these dependencies with `npm install`.

Installing the Dependencies

Node.js provides modules for tasks like accessing the filesystem's input and output, using networking protocols (such as DNS, HTTP, TCP, TLS/SSL, and UDP), and handling binary data. It also provides cryptography modules, interfaces for working with data streams, and much more.

Running `npm install <package>` downloads and places a specific package in the `node_modules` folder, next to your `package.json` file, and adds it to the dependencies list in `package.json`. You should use it whenever you need to add a new module that is required to run the application.

Say you want to create a new Express.js-based server. You'll need to install the Express.js package from <https://npmjs.com>. Here we install a particular version, but to install the latest version, omit the version number and use `npm install express` instead:

```
$ npm install express@4.18.2
added 57 packages, and audited 58 packages in 1s
found 0 vulnerabilities
```

Now the `node_modules` folder contains an `express` folder and additional folders with its dependencies. Also, Express.js is listed as a dependency in `package.json`, as shown in Listing 1-4.

```
{
  "name": " sample-express",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

Listing 1-4: The default `package.json` file with Express.js as a dependency

We've successfully added Express.js as a dependency.

Installing the Development Dependencies

Now let's say you want to use a package called *karma* for end-to-end testing of the server. Instead of being a dependency like Express.js, this package is used only during development and is not necessary for running the actual application.

In cases like this, you should run `npm install --save-dev package` to download this package and add it to the `devDependencies` list in the local `package.json` file:

```
$ npm install --save-dev karma@5.0.0
added 128 packages, and audited 186 packages in 3m
9 vulnerabilities (1 moderate, 4 high, 4 critical)
```

To address issues that do not require attention, run:

```
npm audit fix
```

To address all issues (including breaking changes), run:
`npm audit fix --force`

Run ``npm audit`` for details.

Notice that, after installing the *karma* package, npm indicates that this version has known vulnerabilities. Nonetheless, it is added to the *node_modules* folder and listed as a devDependency in *package.json*. We will follow the suggestions to fix the issues in a bit.

Auditing the package.json File

During installation, npm indicated that *karma* has a vulnerability, so let's verify this. The `npm audit` command inspects the local *package.json* file for any known vulnerabilities:

```
$ npm audit
```

```
# npm audit report
--snip--
karma <=6.3.15
Severity: high
Open redirect in karma - https://github.com/advisories/GHSA-rc3x-jf5g-xvc5
Cross-site Scripting in karma - https://github.com/advisories/GHSA-7x7c-qm48-pq9c
Depends on vulnerable versions of log4js
Depends on vulnerable versions of ua-parser-js
fix available via `npm audit fix --force`
Will install karma@6.4.1, which is a breaking change
--snip--
9 vulnerabilities (1 moderate, 4 high, 4 critical)
```

To address issues that do not require attention, run:
`npm audit fix`

To address all issues (including breaking changes), run:
`npm audit fix --force`

Running the command returns a detailed report about the version and severity of each problematic package, as well as a summary of all the issues found in the currently installed Node.js modules.

The npm package manager also indicated that the issues could be fixed automatically with `npm audit fix`. Alas, it warns us about breaking changes in the latest *karma* version. To accommodate those, we need to use the `--force` flag. I recommend using `npm audit` every few months, along with `npm update`, to avoid using outdated dependencies and creating security risks:

```
$ npm audit fix --force
```

```
added 13 packages, removed 41 packages, changed 27 packages, and audited 158 packages in 5s
```

Now we see that the devDependencies list in *package.json* has the latest *karma* version, and another run of `npm audit` reports that there are no more known vulnerabilities in the installed packages.

Cleaning Up the node_modules Folder

Running `npm prune` inspects the local *package.json* file, compares it to the local *node_modules* folder, and removes all unnecessary packages. You should use it during development, after adding and removing packages, or when performing general cleanup chores.

Let's check that the audit we just performed didn't install any unnecessary packages:

```
$ npm prune
up to date, audited 136 packages in 1s

found 0 vulnerabilities
```

The output looks fine; there are no issues with our packages.

Updating All Packages

Running `npm update` updates all installed packages to their latest acceptable version. You should use this command frequently to avoid outdated dependencies and security risks:

```
$ npm update
added 1 package, removed 1 package, changed 1 package, and audited 158 packages in 8s

found 0 vulnerabilities
```

As you can see, `npm update` displays a summary of the updates.

Removing a Dependency

Running `npm uninstall package` removes the package and its dependencies from the local *node_modules* folder and *package.json* file. You should use this command to delete modules you don't need anymore. Say you decide that end-to-end tests with *karma* are no longer necessary:

```
$ npm uninstall karma
removed 71 packages, and audited 138 packages in 3s

found 0 vulnerabilities
```

The command's output shows the changes made to the *node_modules* folder. The package was removed from *package.json* as well.

Installing a Dependency

Running `npm install` downloads all dependencies and devDependencies from the npm repository and places them in the *node_modules* folder. Use this command to install an existing project on a new machine. For example, to install a copy of the Express.js project in a new folder, you could create a new empty folder and copy only the *package.json* and

package-lock.json files into it. Then you could run the `npm install` command inside this folder:

```
$ npm install
added 137 packages, and audited 138 packages in 3s

found 0 vulnerabilities
```

Whenever you clone the repository or create a new project from a *package.json* file, run `npm install`. As with all previous commands, `npm` shows a status report listing any vulnerabilities.

Using *npx* to Execute a Script Only Once

When you installed Node.js, you also installed `npx`, which stands for *node package execute*. This tool enables you to execute any package from the registry without installing it beforehand. This is useful when you need to run some code only once. For example, you might use a scaffolding script that initializes a project but is neither a dependency nor a development dependency.

The `npx` tool works by checking whether the executable you're trying to run is available through the `$PATH` environment variable or local project binaries. If this is not the case, `npx` installs the package to a central cache instead of your local *node_modules* folder. Say you want to check your *package.json* for syntax errors. For this, you can use the *jsonlint* package. As this package is neither required to run the project nor part of your development process, you don't want to install it into your *node_modules* folder:

```
$ npx jsonlint package.json
Need to install the following packages:
  jsonlint
Ok to proceed? (y) y
{
  "name": " sample-express",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "express": "^4.18.2"
  }
}
```

This calls *jsonlint* to validate our *package.json* file via `npx`. First `npx` installs the package into the global cache folder, then runs *jsonlint*. It prints the content of our *package.json* file and reports no errors. Check your *node_modules* folder; *jsonlint* shouldn't be installed. Nonetheless, on each subsequent call of `npx`, you'll find *jsonlint* available.

Exercise 1: Build a “Hello World” Express.js Server

Express.js is a free and open source backend framework built on top of Node.js. Designed for building web applications and APIs, it is the de facto standard server framework for the Node.js ecosystem and is foundational to full-stack web development.

Express.js offers common middleware used by HTTP servers for tasks such as caching, content negotiation, cookie handling, handling cross-origin requests, redirecting, and much more.

NOTE

Next.js uses its own built-in server that borrows heavily from Express.js. For the Food Finder application that you’ll build in Part II of this book, Next.js will be the foundation of the middleware you’ll use. As Next.js abstracts this middleware for you, you won’t directly interact with the server there.

Let’s build a simple Express.js-based Node.js server to practice your Node.js skills.

Setting Up

If you’ve already created the *sample-express* folder and *package.json* file while following along with this chapter, you can skip this setup. Otherwise, create and switch to a new folder called *sample-express*. Then, to initialize a new Node.js project, run `npm init` on the command line. The interactive guide should ask you for some details, such as the name and version of your application. Accept the defaults for now.

Next, you’ll want to use the Express.js package as the foundation of the server. Run `npm install express@4` to install the latest release of the major version 4. You will see that the *package.json* file now contains *express* as a dependency.

Writing the Server Code

Create an *index.js* file in the *sample-express* folder and add the code in Listing 1-5.

```
const express = require('express');
const server = express();
const port = 3000;

server.get('/hello', function (req, res) {
  res.send('Hello World!');
});

server.listen(port, function () {
  console.log('Listening on ' + port);
});
```

Listing 1-5: A basic Express.js server

First we load the *express* package into the file, instantiate the app, and define a constant for the port to use. Then we create a route for our server so that it will respond to every GET request sent to the */hello* base URL with Hello World! We use the Express.js *get* method and pass */hello* as the first parameter and a callback function as the second parameter. Now, for each GET request sent to the */hello* endpoint, the server runs the callback function that sends Hello World! as the response to the browser. Finally, we use the Express.js *listen* method to spin up the web server and tell it to listen on port 3000.

Start the server from your command line:

```
$ node index.js  
Listening on 3000
```

Now visit *http://localhost:3000/hello* in your browser. You should see the Hello World! message. Congratulations! You just wrote your first Node.js web server in JavaScript.

Summary

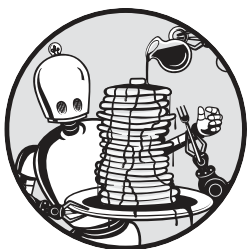
This chapter taught you how to run JavaScript code outside a browser using Node.js and its module ecosystem. You learned how to use, add, and remove modules in your full-stack application with *npm* commands, as well as how to read and use the *package.json* and *package-lock.json* files. Finally, you peeked into Express.js, the de facto standard server for full-stack development, and used it to build a sample Node.js server with just a few lines of code.

This chapter only scratched the surface of Node.js. If you want to explore its full potential, I recommend the Node.js tutorials from W3Schools at <https://www.w3schools.com/nodejs/> and the free ExpressJS Fundamentals course from <https://www.udemy.com/course/expressjs-fundamentals/>.

In the next chapter, you will get to know ES.Next, the latest iteration of JavaScript, and master the modern features it brings to the table.

2

MODERN JAVASCRIPT



In Chapter 1, you used basic JavaScript to create a web server with Node.js. Now we'll take a closer look at the language's more advanced features and how you can effectively use them to create full-stack web applications.

You'll sometimes hear the term *ES.Next* used to refer to new versions of JavaScript. In this book, we use ES.Next as a broad label for modern JavaScript and its concepts. Most runtime environments have implemented the features covered here. Otherwise, you can transpile them with Babel.js, creating backward-compatible JavaScript that emulates the new features for older runtimes.

ES.Next Modules

ES.Next modules allow you to separate code into files to improve maintenance and testability. They encapsulate a piece of logic into easily reusable code, and because variables and functions are limited to the module's

scope, you can use the same variable name in different modules without running into conflicts. (We discuss the concept of scopes in “Declaring Variables” on page 17.)

The official ES.Next modules replaced various unofficial module formats, such as UMD and AMD, which you would load with a `require` statement. For example, you used `require` to include the Express.js package for the Node.js server code in Chapter 1. Instead, ES.Next modules use `export` and `import` statements to export functions from one module’s scope and import them for use somewhere else. In other words, modules allow you to create functions and variables and expose them to a new scope.

Using Named and Default Exports

There are two kinds of Next.js exports: *named* and *default*. These exports use slightly different syntaxes when you import them later. Default exports require you to define new function names on import. For named exports, renaming is optional and done with the `as` statement.

It’s considered a best practice to use named exports over default exports, because named exports define a clear and unique interface for the module’s functionality. When we use default exports, the user risks importing the same function under different names. TypeScript, which we’ll cover in Chapter 3, recommends that we use default exports if the module has one clear purpose and a single export. In contrast, it recommends using named exports whenever the module exports more than one item.

You should know the syntax of default exports so that you can work with third-party modules that use them. Unlike named exports, there can be only one default export per file, marked by the `default` keyword (Listing 2-1).

```
const getFoo = function () {  
  return 'foo';  
};  
  
export default getFoo;
```

Listing 2-1: Default exports

In this listing, we define an anonymous function and store it in the constant `getFoo`. Then we export the constant with the `default` keyword to make it the module’s default export.

You can export named exports inline or at the end of the file, with curly brackets (`{}`). Listing 2-2 shows several named exports.

```
export const getFooBar = function () {  
  return 'foo bar';  
};  
  
const getBar = function () {  
  return 'bar';  
};
```

```
const getBaz = function () {  
    return 'baz';  
};  
  
export {getBar, getBaz};
```

Listing 2-2: Named exports

Here we define an anonymous function, store it in the constant `getFooBar`, and immediately export it as `getFooBar`. Then we define two more anonymous functions and export them as named exports in curly brackets.

Importing Modules

The syntax to import an ES.Next module depends on the type of export you created. Named exports do need to be imported using curly brackets, whereas default exports do not. In Listing 2-3, we import the default export `getFoo` by using the `import` statement followed by the local name we assign to it. Finally, we conclude the import with a reference to the file that contains the code.

```
import getFoo from "default.js";
```

Listing 2-3: Importing default exports

We follow a similar pattern for the named exports in Listing 2-4, except that we need to refer to the original function names inside curly brackets. To rename the functions locally, we would need to explicitly do so with an `as` statement, and there is usually no reason to do so.

```
import { getFooBar, getBar, getBaz } from "named.js";
```

Listing 2-4: Importing named exports

Now you can use the imported functions in your code, as they are available in the scope to which you imported them.

Declaring Variables

JavaScript offers three different ways to declare a variable: `var`, `let`, and `const`. This section discusses the use cases for each of them. Often, you'll be given the advice to avoid `var` because it is "outdated." You can rest assured that it's not, and you must understand each of these variable declaration methods in order to choose the right tool for the job.

These variables differ in their *scope*, which defines the code area in which we can access and use them. JavaScript has multiple levels of scope: global, module, function, and block. *Block* scope, which applies to any block of code enclosed in curly brackets, is the smallest unit of scope. Every time you use curly brackets, you create a new block scope. In comparison, you make a *function* scope when you define a function.

The scope is limited to the code area inside a specific function. The *module* scope applies only to a specific module, whereas the *global* scope applies to the entire program. Variables defined in the global scope are available in every part of your code.

As you'll see in the following code listings, a variable is always available in its own scope and all of its child scopes. Hence, you should remember that, for example, a function scope can contain multiple block scopes. The same variable name can be defined twice in one program as long as each variable occurs in different scopes.

Hoisted Variables

Traditional JavaScript declares variables with the `var` keyword. The scope of these variables is the current execution context (usually the enclosing function). If declared outside any function, the variable's scope is global, and the variable creates a property on the global object.

Unlike for all other variables, the runtime environment moves, or *hoists*, the declaration of `var` to the top of its scope upon execution. Therefore, you can call these variables in your code before you define them. Listing 2-5 shows a short example of hoisting.

```
function scope() {  
    foo = 1;  
    var foo;  
}
```

Listing 2-5: Using a hoisted variable before it is defined

In this listing, we assign a value to a variable before declaring it in the following line. In languages like Java and C, we can't use variables before we declare them, and any attempt to do so will throw an error. However, because of hoisting in JavaScript, the parser moves all variable declarations defined with the `var` keyword to the top of the scope. Thus, the code is equivalent to that in Listing 2-6.

```
function scope() {  
    var foo;  
    foo = 1;  
}
```

Listing 2-6: Defining a variable before using it

Because of hoisting, block scope does not apply to variables declared with the `var` keyword. They are always hoisted. To illustrate this, take a look at Listing 2-7, where we declare a global variable `globalVar`, a variable `foo` inside the function scope, and a variable `bar` inside a block scope, all with the `var` keyword.

```
var globalVar = "global";  
function scope() {  
    var foo = "1";
```

```
    if (true) {
      var bar = "2";
    }
    console.log(globalVar);
    console.log(window.globalVar);
    console.log(foo);
    console.log(bar);
  }
  scope();
```

Listing 2-7: The scope of var

We run the scope function and see that `globalVar` and `window.globalVar` are the same; the parser hoists both variables, `foo` and `bar`, to the top of the function scope. Thus, the variable `bar` is available outside the block scope, and the function scope writes both variables' values, 1 and 2, to the console.

Scope-Abiding Variables

Modern JavaScript introduced the `let` keyword to supplement `var`. With `let`, we can declare variables that are block-scoped and can be accessed only after they have been declared. For this reason, they are considered *non-hoisted* variables. Block-scoped variables are limited to the scope of the block statement inside which they are defined. Unlike global variables defined with `var`, a global `let` variable isn't added to the `window` object.

Let's look at the scope of a variable declared with `let`. In Listing 2-8, we declare a variable `foo` inside a function scope, a variable `bar` inside a block scope, and a global variable `globalVar`.

```
let globalVar = "global";
function scope() {
  let foo = "1";
  if (true) {
    let bar = "2";
  }
  console.log(globalVar);
  console.log(window.globalVar);
  console.log(foo);
  console.log(bar);
}
scope();
```

Listing 2-8: The scope of let

Each variable is available only in its respective scope. The parser does not hoist them, and therefore, the variable `bar` is not available outside the block statement. If you try to reference it elsewhere, the parser will throw an error and notify you that `bar` is not defined.

We execute the function, and unlike the `var` code, it writes only the value of `foo` to the console. When we try to access `bar`, we receive an error, `Uncaught ReferenceError: bar is not defined`. For `globalVar`, we see the value `global` printed on the console, whereas `window.globalVar` is undefined.

Constant-Like Data

Modern JavaScript introduced another new keyword, `const`, for declaring constants such as data types. Like `let`, `const` does not create properties of the global object when declared globally. They, too, are considered non-hoisted, as they cannot be accessed before being declared.

Constants in JavaScript are different from those in many other languages, where they function as immutable data types. In JavaScript, constants only *look* immutable. In fact, they are read-only references to their value. Therefore, you cannot directly reassign another value to the variable identifier for primitive data types. However, objects or arrays are non-primitive data types, so even when you use `const`, you can mutate their values through methods or direct property access.

In Listing 2-9, we declare both a primitive and a non-primitive data type with the `const` keyword and try to change their content.

```
const primitiveDataType = 1;
try {
  primitiveDataType = 2;
} catch (err) {
  console.log(err);
}

const nonPrimitiveDataType = [];
nonPrimitiveDataType.push(1);

console.log(nonPrimitiveDataType);
```

Listing 2-9: Using `const` to declare primitive and non-primitive types

We declare and assign a value to two constant-like data structures. Now when we try to reassign a value to the primitive data structure, the runtime throws the error `Attempted to assign to readonly property`. Because we used `const`, we cannot reassign its value. In contrast, we can modify the `nonPrimitiveDataType` array (done here with the `push` method) and append a value without running into an error. The array should now contain one item with the value 1; hence, we see `[1]` in the console.

Arrow Functions

Modern JavaScript introduced arrow functions as alternatives to regular functions. There are two concepts you need to know about arrow functions. First, they use a different syntax than regular functions. Defining an arrow function is much quicker, requiring just a few characters and one line of code. The second important, but not so obvious, change is that they use something called a lexical scope, making them more intuitive and less error prone.

Writing Arrow Functions

Instead of using the `function` keyword to declare an arrow function, we use the equal-to and greater-than signs to form an arrow (`=>`). This syntax, also called the *fat arrow*, reduces noise and results in more compact code. Therefore, modern JavaScript prefers this syntax when passing functions as arguments.

In addition, if an arrow function has only one parameter and one statement, we can omit the curly brackets and the `return` keyword. In this compact form, we call the function a *concise body* function. Listing 2-10 shows the definition of a traditional function followed by an arrow function.

```
const traditional = function (x) {  
    return x * x;  
}  
  
const conciseBody = x => x * x;
```

Listing 2-10: A traditional function and an arrow function with the concise body syntax

We first define a standard function with the `function` keyword and familiar `return` statement. Then we write the same functionality as an arrow function with the concise body syntax. Here we omit the curly brackets and use an implied `return` statement, without the `return` keyword.

Understanding Lexical Scope

Unlike regular functions, arrow functions do not bind their scope to the object that calls the function. Instead, they use a *lexical scope*, in which the surrounding scope determines the value of the `this` keyword. Therefore, the scope to which `this` refers in an arrow function always represents the object *defining* the arrow function instead of the object *calling* the function. Listing 2-11 illustrates the concepts of lexical and defining scopes.

```
❶ this.scope = "lexical scope";  
  
const scopeOf = {  
    ❷ scope: "defining scope",  
  
    traditional: function () {  
        ❸ return this.scope;  
    },  
  
    arrow: () => {  
        return this.scope;  
    },  
};  
  
console.log(scopeOf.traditional());  
console.log(scopeOf.arrow());
```

Listing 2-11: An arrow function's scope

We first declare the `scope` property on the lexical scope ❶; this is the defining object. Then we create an object with a property of the same name inside the defining scope ❷. Next, we define two functions, both of which use `this` to return the value of `this.scope` ❸.

Upon calling them, you can see the difference between the two references. Whereas `this.scope` in the arrow function refers to the property defined in the lexical scope, the traditional function's `this` refers to the second property we defined. Consequently, the `scopeOf.traditional` function outputs `defining scope`, whereas the `scopeOf.arrow` function outputs `lexical scope`.

Exploring Practical Use Cases

Because functions are first-class citizens in JavaScript, we can pass them as arguments to other functions. In Chapter 1, you used this pattern to define callbacks in Node.js or previously when you worked with event handlers in the browser. But when you use regular functions as callbacks, the code quickly gets cluttered with function statements and curly brackets, even if the actual code in the callback is quite simple. Arrow functions allow for a clean and simple syntax in callbacks. In Listing 2-12, we use a callback on the array `filter` method and define it as a traditional function and as an arrow function.

```
let numbers = [-2, -1, 0, 1, 2];

let traditional = numbers.filter(function(num) {
  return num >= 0;
});

let arrow = numbers.filter(num => num >= 0);

console.log(traditional);
console.log(arrow);
```

Listing 2-12: Passing a fat arrow function as a parameter

The first version of the callback is a traditional function, whereas the second implementation uses an arrow function with a concise body syntax. Both return the same array: `[0, 1, 2]`. We see that the actual functionality, to remove all negative numbers from the array, is a simple check to see if the current item is greater than or equal to zero. The traditional function is harder to understand, as it requires additional characters. Once you fully grasp the arrow syntax, you'll enhance the readability of your code and, in turn, improve the code quality.

Creating Strings

Modern JavaScript introduces untagged and tagged template literals. *Template literals* are a simple way to add variables and expressions to a string. This string

interpolation can span multiple lines and include single and double quotation marks without requiring escaping. We enclose template literals in backticks (``) and indicate a variable or expression in the template by using a dollar sign (\$) and curly brackets.

An *untagged* template literal is just a string enclosed in backticks. The parser interpolates the variables and expressions and returns a string. As a full-stack developer, you'll use this pattern every time you want to add variables to a string or concatenate multiple strings. Listing 2-13 shows an example of an untagged template literal. They can span multiple lines without the need for any control characters.

```
let a = 1;
let b = 2;
let string = `${a} + ${b} = ${a + b}`;
console.log(string);
```

Listing 2-13: An untagged template literal

The parser will substitute the placeholders and evaluate the expression in the template literal to the string `1 + 2 = 3`.

As soon as an expression precedes a template literal, it becomes *tagged*. In these cases, the function receives both a template literal and the substitution values as arguments and then performs an action with both of them before returning a value. This returned value can be of any primitive or non-primitive type. In Listing 2-14, we use a tagged template literal with a custom function to add or subtract numbers and explain the process using words.

```
function tag(literal, ...values) {
  console.log("literal", literal);
  console.log("values", values);

  let result;
  switch (literal[1]) {
    case " plus ":
      result = values[0] + values[1];
      break;
    case " minus ":
      result = values[0] - values[1];
      break;
  }
  return `${values[0]}${literal[1]}${values[1]} is ${result}`;
}

let a = 1;
let b = 2;
let output = tag`What is ${a} plus ${b}?`;

console.log(output);
```

Listing 2-14: A basic tagged template literal

Here the parser calls the tag expression and then passes the template literal and substitution values as arguments to the function. The function constructs a string from the parameters and returns it.

Let's take a deeper look at our code. In our tag expression, the first argument, `literal`, is an array that is split at the variables, like this: ['What is ', ' minus ', '?']. The argument value is also an array, and it contains the values of the template literal variables we passed to the function: [1, 2]. We use a simple switch/case statement to calculate the result based on the literal and values. Finally, we return a new string with the answer to the “question” and see 1 plus 2 is 3 on the console.

With their simple interface for complex string substitutions, tagged template literals provide an elegant way to create a *domain-specific language (DSL)* in JavaScript. A DSL is a language targeted to solve a particular task in a particular domain. It's in contrast to a general-purpose language, such as JavaScript, which we can use to solve a wide array of software-related problems. A familiar example of a DSL is HTML, which we use in the web development domain to mark up text but which we cannot use for mathematical operations or reading file contents. You will define your own DSL for full-stack development with GraphQL schemas. When you define your first GraphQL schema in Chapter 6, you'll understand that its DSL is nothing more than a tagged template literal.

Asynchronous Scripts

JavaScript is single-threaded, which means that it can run only one task at a time. Therefore, long-running tasks can block the application. A simple solution is *asynchronous* programming, a pattern where you start a long-running task without blocking the whole application. While your script waits for a result, the rest of the application can still respond to interactions or user interface events and perform other calculations.

Avoiding Traditional Callbacks

Traditional JavaScript implements asynchronous code with callback functions executed after another function returns a result. You've probably already used callbacks when your code has needed to react to an event instead of running immediately. One common use case for this technique in full-stack web development is performing I/O operations in Node.js or calling remote APIs. Listing 2-15 provides an example of an I/O operation. We import the Node.js `fs` module, which handles filesystem operations, and use a callback function to display the file's contents as soon as the operation concludes.

```
const fs = require("fs");

const callback = (err, data) => {
  if (err) {
    return console.log("error");
  }
}
```

```
    console.log(`File content ${ data }`);  
  };  
  
  fs.readFile(" file.txt", callback);
```

Listing 2-15: Reading a file in Node.js with a callback function

Reading a file is a common example of asynchronous scripting. We don't want the application to be blocked while waiting for the file content to be ready; however, we also need to use the file's content in a specific part of the application.

Here we create the callback function and pass it as a parameter to the `fs.readFile` function. This function reads a file from the filesystem and executes the callback as soon as the I/O operation fails or succeeds. The callback receives the file data and an optional error object, which we write to the console for now.

Callbacks are a clumsy solution to asynchronous scripting. As soon as you have multiple dependent callback functions, you end up in so-called callback hell, where every callback function takes another callback function as an argument. The result is a pyramid of functions that are difficult to read and prone to errors. Modern JavaScript introduced promises and `async/await` as an alternative to callbacks.

Using Promises

Promises provide a much cleaner syntax for chainable asynchronous tasks. Similar to callbacks, they defer further tasks until a previous action has completed or failed. Essentially, promises are function calls that do not return an immediate result. Instead, they promise to return the result at some later point. If there is an error, the promise is rejected instead of resolved.

The Promise object has two properties: the state and the result. When the state is pending, the result is undefined. However, as soon as the promise resolves, the state changes to fulfilled, and the result reflects the return value. If the promise is rejected instead, the state is also set to rejected, and the result contains an error object.

Promises follow a unique syntax. To use them, you first create a new Promise or call a function that returns a Promise. Then you consume the Promise object, and finally you clean up. This is done by registering the consuming functions then, catch, and finally. The promise initially calls then as soon as the state changes from pending to fulfilled and passes the returned data to it. Each following then method receives the return value of the previous one, allowing you to create a single task chain that works with and manipulates these return values.

The promise chain invokes the catch method only if an error occurs either initially or later in the chain of tasks. In addition, a state change (of this particular promise) to rejected also invokes it. In any case, the parser calls the finally method after the stack of then methods has completed or the catch method was invoked. You use the finally method for cleanup tasks

such as unlocking the user interface or closing database connections. It's similar to the `finally` call of a `try...catch` statement.

You can use promises in any function. In Listing 2-16, we use the native `fetch` API to request JSON data.

```
function fetchData(url) {
  fetch(url)
    .then((response) => response.json())
    .then((json) => console.log(json))
    .catch((error) => {
      console.error(`Error : ${error}`);
    });
}
fetchData("https://www.usemodernfullstack.dev/api/v1/users");
```

Listing 2-16: Fetching remote data with promises

Like I/O operations on the filesystem, network requests are long-running tasks that block the application. Therefore, we should use asynchronous patterns to load remote datasets. As in Listing 2-15, we need to wait until the operation is complete before we can process the requested data or handle an error.

The `fetch` API is promise-based by default. As soon as the promise resolves and the state changes to fulfilled, the following `then` function receives the response object. We then parse the data and pass the JSON object to the next function in the *promise chain*, a sequence of functions connected with a dot (`.then`). If there is an error, the promise is rejected. In this case, we catch the error and write it to the console.

Simplifying Asynchronous Scripts

Modern JavaScript introduces a new, simpler pattern for handling asynchronous requests: the `async/await` keywords. Instead of relying on chained functions, we can write code whose structure is similar to regular synchronous code by employing these keywords.

When using this pattern, you mark functions explicitly as asynchronous with `async`. Then you use `await` instead of the promise-based syntax for your asynchronous code. In Listing 2-17, we use the native `fetch` API with `async/await` to perform another long-running task and fetch JSON data from a remote location. This code is functionally the same as Listing 2-16, and you should see that its syntax is more intuitive and cleaner than the chain of `then` calls.

```
async function fetchData (url) {
  try {
    const response = await fetch(url);
    const json = await response.json();
    console.log(json);
  }
```

```
    } catch (error) {  
      console.error(`Error : ${error}`);  
    }  
  }  
  
  fetchData("https://www.usemodernfullstack.dev/api/v1/users");
```

Listing 2-17: Fetching remote data with `async/await`

First we declare the function as `async` to enable the `await` keyword inside the function. Then we use `await` to wait for the response of the `fetch` call. Unlike the promise syntax we used before, `await` simplifies the code. It awaits the response object and returns it. Thus, the code block looks similar to regular synchronous code.

This pattern requires us to handle errors manually. Unlike with promises, there is no default reject function. Therefore, we must wrap `await` statements in a `try...catch` block to handle error states gracefully.

Looping Through an Array

Modern JavaScript introduced a whole set of new array functions. The most important one for full-stack web development is `array.map`. It allows us to run a function on each array item and return a new array with the modified items, preserving the original array. Developers commonly use it in React to generate a list or populate JSX with datasets from arrays. You will use this pattern extensively once we introduce React in Chapter 4.

In Listing 2-18, we use `array.map` to iterate over an array of numbers and create an arrow function as a callback.

```
const original = [1,2,3,4];  
const multiplied = original.map((item) => item * 10);  
console.log(`original array: ${original}`);  
console.log(`multiplied array: ${multiplied}`);
```

Listing 2-18: Using `array.map` to manipulate each item of an array

We iterate over the array items and pass each of them to the callback function. Here we multiply each item by 10, and then `array.map` returns an array with the multiplied items.

When we log the initial array and the returned array, we see that the original array still contains the actual, unchanged numbers (1,2,3,4). Only the multiplied array contains the new, modified items (10,20,30,40).

Dispersing Arrays and Objects

Modern JavaScript's spread operator is written as three dots (`...`). It *spreads out*, or expands, the values of an array or the properties of an object into their own variables or constants.

Technically, the spread operator copies its content to variables that allocate their own memory. In Listing 2-19, we use the spread operator to assign the multiple values of an object to several constants. You'll use this pattern in nearly all React code to access component properties.

```
let object = { fruit: "apple", color: "green" };
let { fruit, color } = { ...object };

console.log(`fruit: ${fruit}, color: ${color}`);

color = "red";
console.log(`object.color: ${object.color}, color: ${color}`);
```

Listing 2-19: Dispersing an object into constants with the spread operator

We first create an object with two properties, `fruit` and `color`. Then we use the spread operator to expand the object into variables and log them to the console. The variables' names are the same as the object properties' names. However, we can now access the values directly from the variables instead of referring to the object. We do so in the template literal and see `fruit: apple, color: green` as the console output.

Also, as these variables allocate their own memory, they are complete clones. Therefore, modifying the variable `color` to `red` won't change the original value: `object.color` still returns `green` when we log both variables to the console.

Using the spread operator to clone an array or object is useful because JavaScript treats arrays as references to its values. When you assign an array or object to a new variable or constant, this merely copies the reference to the original; it does not clone the array or object by allocating memory. Therefore, changing the copy also changes the original. Using spread instead of the equals operator (`=`) allocates memory and keeps no reference to the original value. Hence, it's an excellent solution for cloning an array or object, as shown in Listing 2-20.

```
let originalArray = [1,2,3];
let clonedArray = [...originalArray];

clonedArray[0] = "one";
clonedArray[1] = "two";
clonedArray[2] = "three";

console.log(`originalArray: ${originalArray}, clonedArray: ${clonedArray}`);
```

Listing 2-20: Cloning an array with the spread operator

Here we use the spread operator to copy the values from the original array to the cloned array in the same operation. Then we modify the cloned array's items. Finally, we write the two arrays to the console and see that the original array differs from the cloned array.

Exercise 2: Extend Express.js with Modern JavaScript

Modern JavaScript provides the tools you need to write clean and efficient code. In Part II, you'll use it in the Food Finder application. For now, let's apply your new knowledge to optimize the simple Express.js server you created in Chapter 1.

Editing the package.json File

We'll replace the server's `require` call with named modules for different routes. To do so, we need to explicitly specify that our project uses native modules. Otherwise, Node.js will throw an error. Modify your `package.json` file so that it looks like Listing 2-21.

```
{
  "name": "sample-express",
  "version": "1.0.0",
  "description": "sample express server",
  "license": "ISC",
  "type": "module",
  "dependencies": {
    "express": "^4.18.2",
    "node-fetch": "^3.2.6"
  },
  "devDependencies": {}
}
```

Listing 2-21: The modified package.json file

Add the property `type` with the value `module`. Also, you'll want to install the `node-fetch` package to make an asynchronous API call in one of your routes. Run `npm install node-fetch` to do so.

Writing an ES.Next Module with Asynchronous Code

Create the file `routes.js` in the `sample-express` folder, next to the `index.js` file, and add the code in Listing 2-22.

```
import fetch from "node-fetch";

const routeHello = () => "Hello World!";

const routeAPINames = async () => {
  const url = "https://www.usemodernfullstack.dev/api/v1/users";
  let data;
  try {
    const response = await fetch(url);
    data = await response.json();
  } catch (err) {
    return err;
  }
  const names = data
    .map((item) => `id: ${item.id}, name: ${item.name}`)
```

```

        .join("<br>");
    return names;
};

export { routeHello, routeAPINames };

```

Listing 2-22: The route module in the routes.js file

First we import the `fetch` module for making asynchronous requests. Then we create the first route, for our existing `/hello` endpoint. Its behavior should be the same as before; using a fat arrow function with a concise body syntax, it returns the string `Hello World!`

Next, we create a route for a new `/api/names` endpoint. This endpoint will add a page to our web server displaying a list of usernames and IDs. But first we explicitly define an `async` function so that we can use the `await` syntax for our `fetch` call. Then we define the API endpoint in a constant and another variable to store asynchronous data. We need to define these before we use them because the `await` calls happen inside a `try...catch` block, and these variables are block-scoped. If we defined them inside the block, we wouldn't be able to use them later.

We call the API and `await` the response data, which we convert to JSON as soon as the call succeeds. The `data` variable now holds an array of objects. We use `array.map` to iterate over the data and create the strings we want to display. Then we join all array items with break tags (`
`) to display them in rows and return the string.

Finally, we export the two routes under their names.

Adding the Modules to the Server

Modify the file `index.js` in the `sample-express` folder to match Listing 2-23. We use native modules for importing the `require` module and the routes we created in Listing 2-22.

```

import { routeHello, routeAPINames } from "./routes.js";
import express from "express";

const server = express();
const port = 3000;

server.get("/hello", function (req, res) {
    const response = routeHello(req, res);
    res.send(response);
});

server.get("/api/names", async function (req, res) {
    let response;
    try {
        response = await routeAPINames(req, res);
    } catch (err) {
        console.log(err);
    }
    res.send(response);
}

```



```
});  
  
server.listen(port, function () {  
  console.log("Listening on " + port);  
});
```

Listing 2-23: The basic Express.js server with modern JavaScript

First we import routes with the syntax for named imports. Then we replace the `require` call for the *express* package with an `import` statement. The `/hello` endpoint we created earlier calls the route we imported, and the server sends `Hello World!` as the response to the browser.

Finally, we create a new endpoint, `/api/names`, that contains asynchronous code. Therefore, we mark the handler as *async* and `await` the route inside a `try...catch` block.

Start the server from your command line:

```
$ node index.js  
Listening on 3000
```

Now visit `http://localhost:3000/api/names` in your browser, as shown in Figure 2-1.



Figure 2-1: The response the browser receives from the Node.js web server

You should see the new list of user IDs and names.

Summary

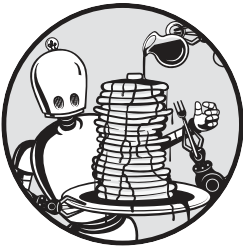
This chapter taught you enough modern JavaScript and ES.Next to create a full-stack application. We covered how to use JavaScript modules to create maintainable packages and import and export code, the different ways to declare variables and constants, the arrow function, and tagged and untagged template literals. We wrote asynchronous code with promises and `async/await`. We also covered `array.map`, the spread operator, and their usefulness for your full-stack code. Finally, you used your new knowledge to update the sample Node.js server from Chapter 1 with modern JavaScript concepts.

Modern JavaScript has many more features than this chapter covers. From the freely available resources, I recommend the JavaScript tutorials at <https://www.javascripttutorial.net>.

In the next chapter, we cover TypeScript, a superset of JavaScript with support for types.

3

TYPESCRIPT



TypeScript is a programming language that adds static typing to the dynamically typed JavaScript language. It's a strict syntactic superset of JavaScript, which means that all existing JavaScript is valid TypeScript. By contrast, TypeScript is not valid JavaScript, because it supplies additional features.

This chapter will introduce you to the pitfalls of working with JavaScript's dynamic types and explain how TypeScript's static typing helps catch errors early, increasing the stability of your code. Full-stack developers have embraced TypeScript: it was the runner-up in the *most wanted* category of a recent Stack Overflow Developer Survey, and 78 percent of participants in a *State of JS* survey reported using it. According to <https://builtwith.com>, TypeScript underlies 7 percent of the top 10,000 sites.

We'll cover the essential and advanced TypeScript concepts necessary for building full-stack applications. Along the way, you'll get to know the

language's most common configuration options, its most important types, and how and when to use TypeScript's static typing features.

Benefits of TypeScript

TypeScript makes working with JavaScript's type system less error prone, as its compiler helps us see type errors instantly. Because JavaScript is *dynamically* typed, you don't need to specify a variable's type when declaring it. As soon as the runtime executes the script, it checks these types based on usage. However, this means that errors resulting from invalid types (for example, calling `array.map` on a variable that holds a number instead of an array) won't be discovered until runtime, at which point the complete program fails.

In addition to being dynamically typed, JavaScript is also *weakly* typed, which means it implicitly converts variables to their most plausible values. Listing 3-1 shows an implicit conversion from a number to a string.

```
let string = "1";
let number = 1;
let result;

result = number + number;
console.log("value: ", result, " type of ", typeof(result));

result = number + string;
console.log("value: ", result, " type of ", typeof(result));
```

Listing 3-1: Implicit conversion from a number to a string in JavaScript

We declare three variables, assigning the first a string, the second a numeric value, and the third the result of using the arithmetic plus (+) operator to add the number to itself. We then log the result of this sum operation and its type to the console. If you executed this code, you would see that the value is numeric and that the runtime assigned a type of `number` to the variable.

Next, we use the same operator again, but instead of adding a numeric value to the `number` variable, we add a string to it. You should see that the logged value is 11, not 2, as you might have expected. Moreover, the variable's assigned type has changed to `string`. This happens because the runtime environment needs to handle an impossible task: adding a number and a string. It solves this issue by implicitly converting the number to a string, then using the plus operator to concatenate the two strings. Without TypeScript, we notice this conversion only when we run the code.

Another common problem caused by untyped variables relates to function and API *contracts*, or the agreements about what the code accepts and returns. When a function takes a parameter, it implicitly expects a parameter of a specific type. But without TypeScript, there is no way to ensure that the parameter type is correct. The same problem exists for the function's return value. To illustrate this, Listing 3-2 changes the code

from Listing 3-1 so that it uses a function to calculate the value of the result variable.

```
let string = "1";
let number = 1;
let result;

const calculate = (a, b) => a + b;

result = calculate(number, number);
console.log("value: ", result, " type of ", typeof(result));

result = calculate(number, string);
console.log("value: ", result, " type of ", typeof(result));
```

Listing 3-2: A function that could return an invalid type due to implicit type conversion

The new calculate function takes two parameters, a and b, and as before, adds the two values. Like in Listing 3-1, as soon as we pass a number and a string as parameters, the function returns a string instead of a number. Our function might expect both parameters to be numbers, but we can't verify this without manually checking the type by using logic similar to that in Listing 3-3.

```
let string = "1";
let number = 1;
let result;

const calculate = (a, b) => {
  if (Number.isInteger(a) === false || Number.isInteger(b) === false) {
    throw new Error("Invalid type: a parameter is not an integer");
  } else {
    return a + b;
  }
};

result = calculate(number, number);
console.log("value: ", result, " type of ", typeof(result));

result = calculate(number, string);
console.log("value: ", result, " type of ", typeof(result));
```

Listing 3-3: The refactored type-safe function

Here we use the native `isInteger` function of the `Number` object to verify that the parameters a and b are integers. The first call of the function, in which we pass it two integers, should calculate the result as expected. The second call, in which we pass the function an integer and a string, looks fine in the editor. However, when we run the code, the runtime environment should throw the error `Invalid type: a parameter is not an integer`.

There are two main concerns with manually checking the types. First, it adds a lot of noise to our code, as we need to check for all possible types every time we work with function or API contracts, such as when we accept a parameter or return a value. Second, we're not notified of issues during

development. To see the errors in dynamically typed languages, we need to execute the code so that the interpreter can inform us about errors at runtime.

Unlike dynamically typed languages, *statically* typed languages perform type checks on the code compilation, before runtime. The TypeScript Compiler (TSC) handles this chore; it can run in the background of our code editor or IDE and instantly report all errors based on invalid type usage. Therefore, you can catch errors and see each variable's assigned types and data structures early.

Even if you don't set up instant feedback like that, running your code through TSC is necessary before it can be used, which ensures that these kinds of errors are caught earlier than they otherwise would be. The ability to check for these errors is one of the most important benefits of using TypeScript over JavaScript. We will discuss how to benefit from type annotations and when to use them in "Type Annotations" on page 38.

Setting Up TypeScript

TypeScript's syntax isn't valid JavaScript, so a regular JavaScript runtime environment can't execute it. To run TypeScript in Node.js or a browser, we first need to use TSC to convert it to regular, backward-compatible JavaScript. We then execute the resulting JavaScript.

Despite being called a compiler, TSC doesn't actually compile TypeScript into JavaScript. Instead, it *transpiles* it. The difference lies in the level of abstraction. A compiler creates low-level code, while a transpiler is a source-to-source compiler that produces equivalent source code in a language of roughly the same abstraction. For example, you could transpile ES.Next to legacy JavaScript or Python 2 to Python 3. (That said, the terms *transpiling* and *compiling* are often used interchangeably.)

In addition to converting TypeScript to JavaScript, TSC checks your code for type errors and verifies the contracts between your functions. The transpiling and type-checking happen independently, and the TSC produces JavaScript regardless of the types you defined. TypeScript errors are merely warnings emitted during the build. They won't stop the transpiling step as long as the JavaScript itself doesn't produce an error.

The use of TypeScript won't affect your code's performance. The compiler removes types and type operations during the transpilation step, essentially stripping all TypeScript syntax from the actual JavaScript code. Therefore, they can't affect the runtime or the size of the final code. TypeScript is consequently no slower than JavaScript, although the transpilation can take some time.

Installation in Node.js

If you're using Node.js, you should define TypeScript and all type definitions as development dependencies with the `--save-dev` flag in your project's *package.json* file. There is no need to install TypeScript globally. Just add TypeScript directly to your project with this npm command:

```
$ npm install --save-dev typescript
```

TypeScript files use the extension `.ts`, and because TypeScript is a superset of JavaScript, all valid JavaScript code is automatically valid TypeScript code. Therefore, you can rename your `.js` files to `.ts` and instantly use the static type checker with your existing code.

A `tsconfig.json` file defines TSC configuration options. We'll cover the most important ones in the next section. For now, run the following command to generate a new file with the default configuration:

```
$ npx tsc -init
```

TSC looks for this file in the current path and all parent directories. The optional `-p` flag points the TypeScript compiler directly to the file. TSC then reads configuration information from this file and treats its folder as TypeScript's root directory.

NOTE

If you want to follow this chapter's examples without creating a dedicated project, you can run code in the online playground at <https://www.typescriptlang.org/play> instead of installing TypeScript locally.

The `tsconfig.json` File

Take a look at the basic structure of a `tsconfig.json` file. The content of the generated file depends on your installed TypeScript version, and there are around 100 configuration properties, but for most projects, only the following few are relevant:

```
{
  "extends": "@tsconfig/recommended/tsconfig.json",
  "compilerOptions": {},
  "include": [],
  "exclude": []
}
```

The `extends` option is a string that configures the path to another similar configuration file. Usually, this property extends a preset you used as a template with minor, project-specific tweaks. It works similarly to class-based inheritance in object-oriented programming. The preset overrides the base configuration, and the configuration's key-value pairs overwrite the preset. The example shown here uses the recommended configuration file for TypeScript to override the default settings.

The `compilerOptions` field configures the transpiling step. We list its options in Appendix A. The value for `include` is an array of strings that specifies the patterns or filenames to include for transpiling. The value for `exclude` is an array of strings that specifies patterns or filenames to exclude. Keep in mind that TSC applies these patterns on the list of files found with the included pattern. Usually, we don't need to include or exclude files, as

our whole project will consist of TypeScript code. Hence, we can leave the arrays empty.

Dynamic Feedback with TypeScript

Most modern code editors have support for TypeScript, and they show us the errors generated by TSC directly inside the code. Remember the `calculate` function we used to explain how TypeScript verifies function contracts? Figure 3-1 is a screenshot from Visual Studio Code highlighting the type error and hinting at the solution.



Figure 3-1: Working with TypeScript in Visual Studio Code

You can use any code editor or IDE you’d like to write your TypeScript code, though one that shows dynamic feedback like this is recommended.

Type Annotations

A type annotation is an optional way to explicitly tell the runtime environment which types to expect. You add them following this schema: *variable: type*. The following example shows a version of the `calculate` function in which we type both parameters as numbers:

```
const calculate = (a: number, b: number) => a + b;
```

Some developers tend to add types to everything in their code, and by doing so, they add noise that makes the code less readable. This anti-pattern, called *over-typing*, stems from a false understanding of how type annotations should work. The TypeScript compiler infers types from usage. Therefore, you don’t need to explicitly type everything. Instead, the code editor runs TSC in the background and leverages the results to display the inferred type information and compiler errors as you saw in the “Dynamic Feedback with TypeScript” section.

Rather, type annotations are a way to ensure that code honors the API contracts. There are three scenarios in which you’ll want to verify the contract, and only one of them is especially important. The first scenario, upon a variable’s declaration, is usually not recommended. The second, annotating the return value of a function, is optional, whereas the third scenario, annotating a function’s parameters, is essential. We’ll now take a look at all three of these cases in detail.

Declaring a Variable

The most obvious place to type a variable is upon an assignment or declaration. Listing 3-4 demonstrates this by explicitly typing the variable `weather` as a string and then assigning it a string value.

```
let weather: string = "sunny";
```

Listing 3-4: Over-typing during the variable's declaration

In most cases, however, this is a form of over-typing, as you could instead leverage the compiler's type inference. Listing 3-5 shows the alternative pattern of using type inference.

```
let weather = "sunny";
```

Listing 3-5: Inferring the variable's type based on its value

Because TSC automatically infers the type of this variable, the code editor should show the type information when you hover over the variable. Without the explicit annotation, we have a much cleaner syntax and avoid the noise that the redundant type declaration adds to the code. This improves code readability, which is why this kind of over-typing is usually to be avoided.

Declaring a Return Value

Although TypeScript can infer a function's return type, you'll usually want to annotate it explicitly. This code pattern ensures that the function's contract is honored, as the compiler shows implementation errors where the function is defined instead of where it is used.

Another reason to use type annotations in this situation is that, as a programmer, you must explicitly define what a function does. By clarifying the function's input and output types, you'll gain a better understanding of what you actually want the function to do. Listing 3-6 shows you how to declare a function's return type upon declaration.

```
function getWeather(): string {  
    const weather = "sunny";  
    return weather;  
}
```

Listing 3-6: Typing a function's return value upon declaration

We create a function that returns the weather variable we declared earlier. The weather variable has the inferred string type. Hence, the function returns a string. Our type definition explicitly sets the function's return type.

Declaring a Function's Parameters

It's essential to annotate the parameters of a function, because TypeScript doesn't have enough information to infer function parameters in most

cases. By typing these parameters, you're telling the compiler to check the types when you call the function and pass it arguments. Take a look at Listing 3-7 to see this pattern in action.

```
const weather = "sunny";
function getWeather(weather: string): string {
    return weather;
};
getWeather(weather);
```

Listing 3-7: Typing a function's parameters

Instead of declaring the weather variable as a constant inside the function, we want the returned value to be dynamic. Therefore, we modify the function to accept a parameter and return it immediately. We then call the function with the weather constant as a parameter.

Good TypeScript code avoids noise and relies on inferring type annotations. It always annotates a function's parameters and opts for annotated return values but never annotates local variables.

Built-in Types

Before you can use TypeScript and its annotations, you need to know what types are available to you. One of TypeScript's main benefits is that it enables you to declare any of JavaScript's primitive types explicitly. In addition, TypeScript adds its own types, the most important of which are unions, tuples, any, and void. You can also define custom types and interfaces.

Primitive JavaScript Types

JavaScript has five primitive types: strings, numbers, Booleans, undefined, and null. Everything else in the language is considered an object. Listing 3-8 shows the syntax for defining variables of these primitive JavaScript types with additional TypeScript type annotations. (Remember that, most of the time, you can just rely on the compiler's type inference in this situation.)

```
let stringType: string = "bar";
let booleanType: boolean = true;
let integerType: number = 1;
let floatType: number = 1.5;
let nullType: null = null;
let undefinedType: undefined = undefined;
```

Listing 3-8: JavaScript's primitive types with TypeScript's type annotations

First we define a string variable and a Boolean with the TypeScript annotations. These are identical to strings and Booleans in JavaScript. Then we define two numbers. Like JavaScript, TypeScript uses a single generic type for numbers, without differentiating between integers and

floating points. Finally, we look at TypeScript's null and undefined types. These behave the same as JavaScript's primitive types of the same name. *Null* refers to a value that either is empty or doesn't exist, and it indicates the intentional absence of a value. In contrast, *undefined* indicates the unintentional absence of a value. We did not assign a value in Listing 3-5 for the undefined type, because we don't know it.

The union Type

There are a few additional types you should know about, because the more precise your type annotations are, the more helpful you'll find TSC to be. TypeScript introduced the union type to the JavaScript ecosystem. *Unions* are variables or parameters that can have more than one data type. Listing 3-9 shows an example of a union type that can be a string or a number.

```
let stringOrNumberUnionType: string | number;
stringOrNumberUnionType = "bar";
stringOrNumberUnionType = 1;
stringOrNumberUnionType = true;
```

Listing 3-9: TypeScript's union type

We declare a union-type variable that can contain either a string or a number, but nothing else. As soon as we assign a Boolean variable, TSC throws an error, and the IDE shows the message Type 'boolean' is not assignable to type 'string | number'.

While you might find union types useful for annotating function parameters and arrays that can contain different types, you should use them sparingly and avoid them whenever possible. This is because, before working with union-typed items, you need to perform additional manual type checks; otherwise, they could cause errors. For example, if you iterated over an array of strings or numbers and then added all items, you would first need to convert all strings to numbers. Otherwise, JavaScript would implicitly convert the numbers to strings, as shown earlier in this chapter.

The array Type

TypeScript provides a generic array type that offers array functions similar to JavaScript's array. However, take a close look at the syntax for typing the array, shown in Listing 3-10. You'll notice that the type of the array depends on the type of the array items.

```
let genericArray: [] = [];
genericArray.push(1);

let numberArray: number[] = [];
numberArray.push(1);
```

Listing 3-10: Typed arrays

First we define an array without specifying the type of its items. Unfortunately, what seems to be a definition of a generic array leads to issues down the road. As soon as we try to add a value, TSC throws the error `Argument of type 'number' is not assignable to parameter of type 'never'`, because the array is not typed.

Hence, we need to type the items in the array. Therefore, we create an array, `numberArray`, in which each item has the type of `number`. Now we can add numeric values to the array without running into errors.

The object Type

TypeScript's built-in object type is the same as JavaScript's object. Although you can define the properties' types for TSC to type-check, the compiler can't ensure the order of the properties. Nonetheless, it typechecks them, as shown in Listing 3-11.

```
let weatherDetail: {  
    weather: string,  
    zipcode: string,  
    temp: number  
} = { weather: "sunny", zipcode: "00000", temp: 1 };  
weatherDetail.weather = 2;
```

Listing 3-11: Typed objects

Here we define an object with three properties: two that take a string and another that takes a number. Then we try to assign a number to the property `weather` annotated as a string. Now TSC notifies us with an error explaining that we assigned a value of the wrong type.

Note that, usually, you should avoid typing objects inline, as in this example. Instead, it is a best practice to create a custom type, which is reusable and avoids cluttering our code, enhancing its readability. We discuss how to create and use them in “Custom Types and Interfaces” on page 44.

The tuple Type

Another common type that TypeScript adds to JavaScript is the tuple type. Shown in Listing 3-12, *tuples* are arrays with a specified number of typed items. TypeScript's tuples are similar to those you might have encountered in programming languages such as Python and C#.

```
let validTuple: [string, number] = ["bar", 1];  
let invalidTuple: [string, number] = [1, "bar"];
```

Listing 3-12: TypeScript's tuple type

We define two tuples. In both, the first array item is a string, and the second is a number. If the type, order, or number of items added to the tuple differs from the tuple's declaration, TSC throws an error. Here the first

assignment is acceptable, whereas the second one throws two errors indicating a mismatch in types.

The any Type

TypeScript's `any` type is generic, meaning it can take any value, and you should avoid using it. As you can see in Listing 3-13, it accepts all values without throwing an error, which defeats the purpose of static typing.

```
let indifferent: any = true;
indifferent = 1;
indifferent = [];
```

Listing 3-13: TypeScript's any type

Using `any` might seem like an easy choice, and it is tempting to rely on it as an escape hatch. Avoid this at all costs. When you pass `any` as a value to, say, a function, you break the contract you specified in the function declaration, and when you use `any` to define the contract, there effectively isn't one.

To view a scenario in which using the `any` type causes problems, take a look at Listing 3-14.

```
const calculate = (a: any, b: any): any => a + b;
console.log(calculate (1,1));
console.log(calculate ("1",1));
```

Listing 3-14: Problems caused by the any type

We reuse the `calculate` function, which adds two numbers. When we pass two numeric values, we receive the expected output of 2. In a previous example, we typed the parameters as numbers, thus preventing the use of invalid types as arguments.

However, when we use `any` instead of a number and pass a string to the function, TSC doesn't throw an error. JavaScript implicitly converts the number to a string and returns an unexpected value of 11. We saw this behavior at the beginning of the chapter, in the untyped version of the function. As you can see, using `any` is the same as using no types at all.

While convenient, the `any` type masks your bugs during programming and hides your type designs, rendering type-checking useless. It also prevents your IDE from displaying errors and invalid types.

The void Type

TypeScript's `void` type is the opposite of `any`: it indicates no type at all. Its only use case is to annotate the return value of a function that shouldn't have one, as shown in Listing 3-15.

```
function log(msg: string): void {  
    console.log(msg);  
}
```

Listing 3-15: TypeScript's void type

The custom log function we define here passes a parameter to the console. It doesn't return anything, so we use `void` as the return type.

To learn more about TypeScript types and other important details of the language, take a look at *The TypeScript Handbook* at <https://www.typescriptlang.org/docs/handbook/intro.html>.

Custom Types and Interfaces

The previous sections introduced you to enough TypeScript to begin using the language. However, you'll find it helpful to know a few more advanced concepts. This section shows you how to create custom types and use untyped third-party libraries in your TypeScript code. You'll also learn when to create a new type and use a custom interface.

While working with TypeScript, remember that a TypeScript file *without* top-level imports or exports is not a module; therefore, it runs in the *global* scope. Consequently, all of its declarations are accessible in other modules. By contrast, a TypeScript file *with* top-level imports or exports is its own module, and all declarations are limited to the *module* scope, meaning they're available in the scope of this module only.

Defining Custom Types

TypeScript lets you define custom types by using the `type` keyword. Custom types are a great way to simplify your code. To see how, take a second look at the code shown back in Listing 3-8, when you created a typed object. Now consider Listing 3-16, which optimizes the code with a custom type definition. You should find it much cleaner and easier to read.

```
type WeatherDetailType = {  
    weather: string;  
    zipcode: string;  
    temp?: number;  
};  
  
let weatherDetail: WeatherDetailType = {  
    weather: "sunny",  
    zipcode: "00000",  
    temp: 30  
};  
  
const getWeatherDetail = (data: WeatherDetailType): WeatherDetailType => data;
```

Listing 3-16: Custom types for typed objects with TypeScript

We create a custom type, `WeatherDetailType`, with the `type` keyword. Note that the overall syntax is similar to that used to define an object; we use the equal sign (`=`) to assign the definition to the custom type.

The custom type has two required properties: `weather` and `zipcode`. In addition, it has an optional `temp` property, as indicated by the question mark (`?`). Now when we create the `getWeatherDetail` function, we can annotate the parameter, `weatherDetail`, as an object with a type of `WeatherDetailType`. Using this technique, we avoid using inline annotations and can reuse our custom type later, such as to annotate the return type of a function.

Defining Interfaces

In addition to types, TypeScript has interfaces. However, the difference between a type and an interface is blurry. You can freely decide which one to use, so long as you follow a convention in your code.

In general, we consider a *type* definition to answer the question, “Which type is this data?” A possible answer might be a union or a tuple. An *interface* is a way to describe the shape of some data, such as the properties of an object. It answers the question, “Which properties does this object have?” The most practical difference is that, unlike an interface, we cannot directly modify a type after we’ve declared it. For an in-depth look at the distinction, consult *The TypeScript Handbook*.

As a rule of thumb, use an interface to define a new object or the method of an object. More generally, consider using interfaces over types, as they provide more precise error messages. A classic React use case for interfaces is to define the properties of a specific component. Listing 3-17 shows how to use the `interface` keyword to create a new interface to replace the type in Listing 3-16.

```
interface WeatherProps {  
  weather: string;  
  zipcode: string;  
  temp?: number;  
}  
  
const weatherComponent = (props: WeatherProps): string => props.weather;
```

Listing 3-17: Custom interfaces for TypeScript functions

Here we use the `interface` keyword to define a new interface. Unlike a custom type’s definition, an interface definition does not use the equal sign to assign the interface’s properties to its name. We then use the custom interface to type the properties object `props` of the `weatherComponent`, which returns a string.

Using Type Declaration Files

To use custom types universally, you can define them in *type declaration files*, which have the `.d.ts` extension. Unlike regular TypeScript files with the `.ts` or `.tsx` extension, type declaration files shouldn’t contain any implementation

code. Instead, TSC uses these type definitions to understand custom types and perform type checks. They aren't transpiled to JavaScript and are never part of the executed script.

Type declaration files prove useful when you find yourself working with external code bases. Often, third-party libraries aren't written in TypeScript. Therefore, they don't provide type declaration files for their code bases. Luckily, the DefinitelyTyped repository at <http://definitelytyped.github.io> provides type declaration files for more than 7,000 libraries. Use these files to add TypeScript support to these libraries.

Type declaration files are collected under the `@types` scope in npm. This scope holds all the declarations from DefinitelyTyped. Hence, they are easy to find and are grouped next to each other in your `package.json` file. All type declaration files from the `@types` scope should be considered development dependencies of your project. Hence, we use the `--save-dev` flag on the npm install command to add them.

Listing 3-18 shows a minimal example of a type declaration file that exports a type and interface for an API.

```
interface WeatherQueryInterface {  
  zipcode: string;  
}  
  
type WeatherDetailType = {  
  weather: string;  
  zipcode: string;  
  temp?: number;  
};
```

Listing 3-18: Defining custom types and interfaces

Save these definitions in a file called `custom.d.ts` in your root directory. TSC should automatically load these definitions. You can now use the types and interfaces from the file in your TypeScript modules.

Exercise 3: Extend Express.js with TypeScript

Let's use your new knowledge of TypeScript to rewrite the Express.js server you created in Exercises 1 and 2. In addition to adding type annotations, we'll add a new route to the server by using custom types.

Setting Up

Begin by adding TypeScript to the project following the steps described in "Setting Up TypeScript" on page 36. Next, because Express.js isn't typed, add type definitions from DefinitelyTyped to your project by running the following:

```
$ npm install --save-dev @types/express
```

Your *package.json* file should now look like this:

```
{
  "name": "sample-express",
  "version": "1.0.0",
  "description": "sample express server",
  "license": "ISC",
  "type": "module",
  "dependencies": {
    "express": "^4.18.2",
    "node-fetch": "^3.2.6"
  },
  "devDependencies": {
    "@types/express": "^4.17.15",
    "typescript": "^4.9.4"
  }
}
```

Now you can create configuration and type declaration files for the project.

Creating the *tsconfig.json* File

Either create a new *tsconfig.json* file in the *sample-express* folder, next to the *index.ts* file, or open the one you created earlier. Then add or replace its content with the following code:

```
{
  "compilerOptions": {
    "esModuleInterop": true,
    "module": "es6",
    "moduleResolution": "node",
    "target": "es6",
    "noImplicitAny": true
  }
}
```

We configure TypeScript for our simple Express.js server, which requires only a few settings. We use ES.Next modules for our TypeScript code, and because we want to keep them after transpiling the TypeScript to JavaScript, we set `module` and `target` to `es6`. The *express* package is a CommonJS module. Therefore, we need to use the `esModuleInterop` option and set the `moduleResolution` to `node`. Finally, we use the `noImplicitAny` option to disallow the implicit use of the `any` type and require explicit typing. Appendix A describes these configuration options in more detail.

Defining Custom Types

For our server, we'll follow a simple rule of thumb: every time we use an object, we should consider adding a custom type or interface to our project. If the object is a function parameter, we'll create a custom interface. If we use this particular object more than once, we'll create a custom type.

To define the custom types for this sample project, we create a file *custom.d.ts* next to the *index.ts* file in the *sample-express* folder and add the code from Listing 3-19.

```
type responseItemType = {
  id: string;
  name: string;
};

type WeatherDetailType = {
  zipcode: string;
  weather: string;
  temp?: number;
};

interface WeatherQueryInterface {
  zipcode: string;
}
```

Listing 3-19: The custom.d.ts file

We create two custom types and an interface. One defines the response items of the asynchronous API call. The other type and the interface are similar to examples shown earlier in this chapter. They are necessary for the new weather route we will create shortly.

Adding Type Annotations to the routes.ts File

Next, we must add type annotations to our server code. Rename the *routes.js* file in the *sample-express* folder to *routes.ts* to enable the TSC for this file. You should instantly see the errors and warnings appear in your editor. Take some time to look at these and then adjust the contents to match the code in Listing 3-20. We've bolded all type annotations.

```
import fetch from "node-fetch";

const routeHello = (): string => "Hello World!";

const routeAPINames = async (): Promise<string> => {
  const url = "https://www.usemodernfullstack.dev/api/v1/users";
  let data: responseItemType[];
  try {
    const response = await fetch(url);
    data = (await response.json()) as responseItemType[];
  } catch (err) {
    return "Error";
  }
  const names = data
    .map((item) => `id: ${item.id}, name: ${item.name}`)
    .join("<br>");
  return names;
};
```

```

const routeWeather = (query: WeatherQueryInterface): WeatherDetailType =>
  queryWeatherData(query);

const queryWeatherData = (query: WeatherQueryInterface): WeatherDetailType => {
  return {
    zipcode: query.zipcode,
    weather: "sunny",
    temp: 35
  };
};

export { routeHello, routeAPINames, routeWeather };

```

Listing 3-20: The typed routes.ts file

Following the principle discussed in “Type Annotations” on page 38, we annotate only a function’s parameters and return types. We also annotate local variables only when their types cannot be inferred, as when converting the fetch response to JSON. Here we need to explicitly type the variable with our custom `responseItemType` and cast the conversion’s return value as an array of `responseItemTypes`.

In the rest of the listing, we create the functions for the additional weather route. We use the custom interface for typing both functions’ parameters and the custom type for their return types. In this basic example, the query function returns mostly static data, except the ZIP code, which it takes from the passed parameters. A regular implementation would query a database with the ZIP code and retrieve actual data.

Finally, we add the new route for the weather endpoint to the export statement.

Adding Type Annotations to the index.ts File

Rename the file `index.js` in the `sample-express` folder to `index.ts` and adjust the code to match Listing 3-20. In addition to the necessary type annotations, create a new endpoint and follow the TypeScript convention to prefix unused parameters with an underscore (`_`), shown in Listing 3-21.

```

import { routeHello, routeAPINames, routeWeather } from "./routes.js";
import express, { Request, Response } from "express";

const server = express();
const port = 3000;

server.get("/hello", function (_req: Request, res: Response): void {
  const response = routeHello();
  res.send(response);
});

server.get("/api/names",
  async function (_req: Request, res: Response): Promise<void> {
    let response: string;
    try {

```

```

        response = await routeAPINames();
        res.send(response);
    } catch (err) {
        console.log(err);
    }
}
);

server.get(
    "/api/weather/:zipcode",
    function (req: Request, res: Response): void {
        const response = routeWeather({ zipcode: req.params.zipcode });
        res.send(response);
    }
);

server.listen(port, function (): void {
    console.log("Listening on " + port);
});

```

Listing 3-21: The typed index.ts file

First we import the new weather route from the available routes and the Request and Response types from the *express* package. These are all named exports. Thus, we use curly brackets (`{}`).

Then, following best practices, we add code annotations and, at the same time, prefix the unused req parameters with an underscore. TSC will follow the convention of functional programming languages by ignoring these parameters. The *api/names* entry point is marked as an async function, so it needs to return a value wrapped in a promise. Hence, nothing is returned, and we return void as the promise's value.

In the following lines of code, we create an additional route for a new */api/weather/:zipcode* endpoint. The colon (`:`) creates a parameter on the request's params object. We retrieve the value for zipcode with `req.params.zipcode` and pass it down to the `routeWeather` function. Note that there is no underscore on the request parameter this time. Finally, we use the same function as before to start the Express.js server and listen to port 3000.

Transpiling and Running the Code

To transpile the code with the TypeScript compiler to JavaScript, run TSC with `npx` on the command line:

```
$ npx tsc
```

TSC generates two new files, *index.js* and *routes.js*, from the TypeScript files. Start the server from your command line with the regular Node.js call:

```
$ node index.js
Listening on 3000
```

Now visit `http://localhost:3000/api/weather/12345` in your browser. You should see the weather details with the ZIP code 12345, as shown in Figure 3-2.



Figure 3-2: Browser response from the Node.js web server

Success! You wrote your first TypeScript application.

Summary

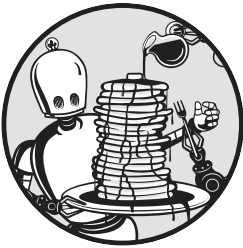
This chapter taught you what you need to know about TypeScript to create a full-stack application. We set up TypeScript and TSC in a new project, then discussed its most important configuration options. Next, you learned to use TypeScript efficiently, leveraging type-annotation inference to avoid over-typing.

We also discussed primitive and advanced built-in types and how to create custom types and interfaces. Finally, you used your new knowledge to add TypeScript to the Express.js server built in previous exercises and refactored the code with type annotations, custom types, and interfaces.

If you want to become a TypeScript expert, I recommend *The TypeScript Handbook* and the tutorials at <https://www.typescripttutorial.net>. In the next chapter, you'll get to know React, a declarative JavaScript library for building user interfaces.

4

REACT



Developers can use the React library to create a full-stack application's user interface. React is built upon the Node.js ecosystem, and as one of the most commonly used web frameworks, it currently forms the basis of more than 40 percent of the most visited websites.

To work effectively with React, you must understand the syntax used to define the appearance of user interface elements and then combine these into React components that can dynamically update. This chapter covers everything you need to know to begin developing full-stack applications using this library.

The Role of React

Modern frontend architectures split an application's user interface into small, self-contained, and reusable items. Some of these, such as headers, navigations, and logos, might appear only once per page, while others are

repeated elements that form the page's contents, such as headlines, buttons, and teasers. Figure 4-1 shows some of these items. React's syntax embraces this pattern; the library focuses on building these independent components and, in doing so, helps us develop our applications more efficiently.

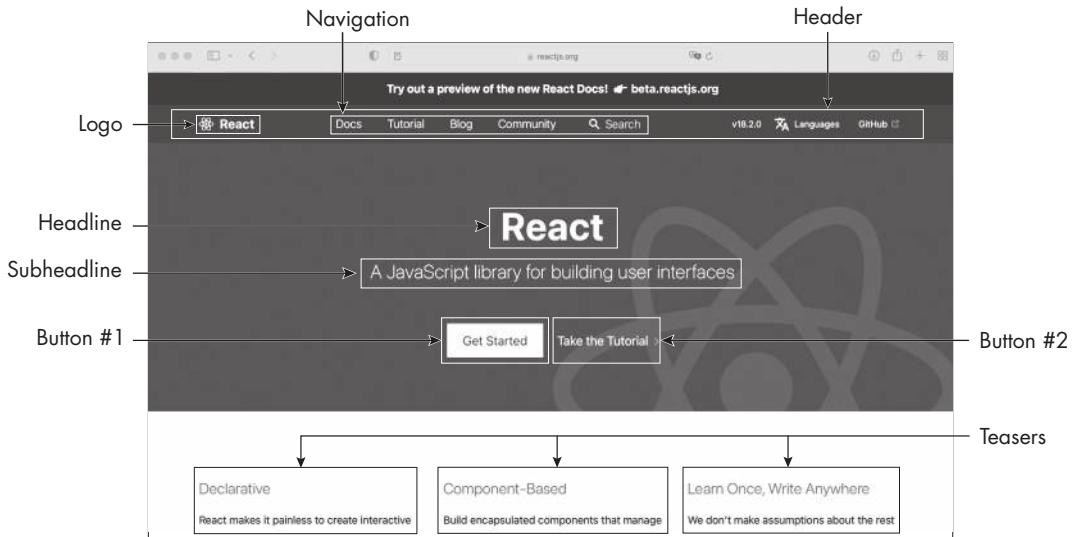


Figure 4-1: User interface components

React uses a *declarative* programming paradigm, through which you create a user interface by describing the desired results instead of explicitly listing all the steps necessary to create it, as is done in *imperative* programming. A classic example of the declarative paradigm is HTML. Using HTML, you describe a web page's elements, and the browser then renders the page. By contrast, you could use JavaScript to write an imperative program that creates each HTML element. In doing so, you would explicitly list the steps to build the website.

In addition, these user interface components are *reactive*. This means two things: one, that they handle their own isolated states, and two, that each component updates the page's HTML as soon as its state changes. Changes to the React code instantly affect a browser's *document object model* (DOM), which represents a website as a tree in which each HTML element is a node. The DOM also provides an API for each node and for the website in general, enabling scripts to modify a website or a specific node.

DOM operations, such as re-rendering a component, are expensive. To update the DOM, React uses a *virtual DOM*, which is an in-memory clone of the actual browser DOM that it later syncs with the real thing. This virtual DOM allows for incremental updates that reduce the number of costly operations on the browser. The virtual DOM is a crucial principle of React. React calculates the difference between the virtual DOM and the real DOM with every call to one of its render functions and then decides what to

update. Usually, React performs batch updates to lower the performance impact further. This process of reconciliation lets React deliver fast and responsive user interfaces.

Although React is primarily a user interface library, developers can also use it to build single-page applications that don't require middleware or a backend. These apps are nothing more than a view layer rendered in the browser. To some extent, they can be dynamic: for example, we can change the page's language, open an image gallery, or toggle an element's visibility. However, all of this occurs in the browser, with additional React modules, rather than on the server.

We can also perform more advanced functionality, like updating the browser's location to simulate the existence of distinct pages, purely in the browser, with React's Router module. This module lets us define routes, similar to the ones we defined in our Express.js server, on the frontend. As soon as a user clicks an internal link, the routing component updates the view and changes the browser's location. This makes it seem as though they've loaded another HTML page. In reality, we've just changed the current page's contents. In doing so, we avoided another set of server requests, so the simulated page loads much more quickly. Also, because our JavaScript code controls the transition between pages, we can add effects and animations to these transitions.

Setting Up React

Unlike, say, the basic Express.js server you created in Exercise 1 on page 13, which uses standard JavaScript and can run directly with Node.js, React relies on an advanced setup with a complete build toolchain. For example, it uses a custom JavaScript Syntax Extension (JSX) to describe HTML elements and TypeScript for static typing, both of which require a transpiler to convert the code to JavaScript. Therefore, the manual process for setting up React is quite complex.

Thus, we generally rely on other tools. In the case of a single-page application, we use a code generator, such as `create-react-app`, to scaffold it. During this scaffolding process, `create-react-app` generates the boilerplate code for a new React application, as well as the build chain and folder structure for the project. It also provides a consistent project layout that helps us easily understand other React projects.

To run the examples in this chapter, one option is to scaffold a simple TypeScript React app with `create-react-app` by following the steps at <https://create-react-app.dev/docs/getting-started/>. If you don't want to create a dedicated project, you can instead run code using React with a TypeScript template in an online playground, such as <https://codesandbox.io> or <https://stackblitz.com>. The playgrounds and `create-react-app` follow the same file structure. In both cases, you should save your code to the default `App.tsx` file.

For more complex apps, we'd use a complete web application framework such as Next.js, which provides the necessary setup out of the box. Covered in Chapter 5, Next.js is the most popular framework for full-stack

web applications that use React. Internally, Next.js employs a variation of create-react-app for scaffolding. We'll rely on it in future chapters to work with React.

The JavaScript Syntax Extension

React uses JSX to define the appearance of user interface components. JSX is an extension of JavaScript that a transpiler must convert before the browser renders it to the DOM. While it has HTML-like syntax, it is more than a simple templating language. Instead, it allows us to use any JavaScript feature to describe React elements. For example, we can use JSX syntax inside conditional statements, assign it to variables, and return it from functions. The compiler will then embed any variable or valid JavaScript expression wrapped in curly brackets (`{}`) into the HTML.

This logic allows us to, for instance, use `array.map` to loop over an array, check each item for a certain condition, pass the item to another function, and create a set of JSX elements based on the function's return value, directly inside a page's template. While this may sound abstract, we'll use this pattern extensively when we create React components in the Food Finder application you'll build in Part II.

An Example JSX Expression

JSX expressions, like those in Listing 4-1, are the most essential part of the React user interfaces. This JavaScript code defines a JSX function expression, `getElement`, that takes one string as a parameter and returns a `JSX.Element`.

```
import React from "react";

export default function App() {
  const getElement = (weather: string): JSX.Element => {
    const element = <h1>The weather is {weather}</h1>;
    return element;
  };
  return getElement("sunny");
}
```

Listing 4-1: A minimal example of a JSX expression

The entry point for each React application is the `App` function. Like the `index.js` file of our Express.js server, this function is executed when the application starts. Here, we usually set up the global elements, such as stylesheets and the overall page layout.

React renders the function's return value to the browser. In Listing 4-1, we immediately return an element. As the smallest building blocks of React user interfaces, *elements* describe what you'll see on the screen, just as HTML elements do. Examples of elements include custom buttons, headlines, and images.

After importing the React package, we create the JSX element and store it in an element constant. At first glance, you might wonder why it isn't wrapped in quotes, as it contains what appears to be a regular HTML `h1` element and looks like a string. The answer is that it isn't a string but a JSX element from which the library creates HTML elements programmatically. As a result, the code will display a message about the weather to the page.

As soon as we call the JSX expression, the React library transpiles it into a regular JavaScript function call and creates an HTML string from the JSX element displayed in the browser. In Chapter 3, you learned that all valid JavaScript is also valid TypeScript. Hence, we can use JSX with TypeScript as well. JSX files use a `.jsx` (JavaScript) or `.tsx` (TypeScript) extension. Paste this code into the `App.tsx` file of the project you created, and the browser should render an `h1` HTML element with the text `The weather is sunny` either in the preview pane of the online playground or in your browser.

The ReactDOM Package

One easy way to work with elements is to use the ReactDOM package, which contains APIs for working with the DOM. Note that the elements you create aren't browser DOM elements. Instead, they're plain JavaScript objects that will be rendered, using React's render function, to the virtual DOM's root element and then attached to the browser DOM.

React elements are *immutable*: once created, they cannot be changed. If you do alter any part of the element, React will create a new element and re-render the virtual DOM, then compare the virtual DOM with the browser DOM to decide whether the browser DOM needs an update. We'll use JSX abstractions for these tasks; nonetheless, it's good to understand how React works under the hood. If you want to dig deeper, consult the official documentation at <https://react.dev/learn>.

Organizing Code into Components

We mentioned that components are independent, reusable pieces of code built from React elements. Elements are objects that can contain other elements. Once rendered to the virtual or browser DOM, they create DOM nodes or whole DOM subtrees. Meanwhile, React *components* are classes or functions that output elements and render them to the virtual DOM. We will build a user interface using React components. For more information about this distinction, read the deep dive at the official React blog: <https://reactjs.org/blog/2015/12/18/react-components-elements-and-instances.html>.

While other frameworks might separate a user interface's code by technology, splitting it into HTML, CSS, and JavaScript files, React instead separates code into these logical building blocks. As a result, a single physical file contains all the information necessary for a component, regardless of underlying technologies.

More concretely, a React component is a JavaScript function that, by convention, starts with an uppercase letter. Furthermore, it takes a single object argument, called `props`, and returns a React element. This `props`

argument should never be modified inside the component and is considered immutable inside the React code.

Listing 4-2 shows a basic React component that displays the same weather string as in the previous listings. In addition, we've added a custom interface and a click handler. The custom interface enables us to set an attribute on the JSX component and read its value in the TypeScript code. It's a common way to pass values to a function component without a global state management library.

Here, we simply pass the component the same string used in the previous listings and render it to the DOM, but for a real-world application, the weather string might be part of an API response. To get the weather data, a parent component might query the API and then send this data through the component's attribute to the component's code, or each component in the application would need to query the API to access that data, impacting the overall performance of the application.

The click handler enables us to react to user interactions. In JSX, click handlers have the same names as in HTML, and we add them the way we might add inline DOM events. For example, to react to a user clicking an element, we add an `onClick` attribute with a callback function.

```
import React from "react";

export default function App() {

  interface WeatherProps {
    weather: string;
  }

  const clickHandler = (text: string): void => {
    alert(text);
  };

  const WeatherComponent = (props: WeatherProps): JSX.Element => {
    const text = `The weather is ${props.weather}`;
    return (<h1 onClick={() => clickHandler(text)}>{text}</h1>);
  };

  return (<WeatherComponent weather="sunny" />);
}
```

Listing 4-2: A basic React component

First we create a custom interface for our new component's properties. We'll use this interface for the component's `prop` parameter later. Because we set a `weather` attribute on the component and define a matching `weather` property on the interface, we can access the value of the `weather` attribute with `props.weather` in our TypeScript code.

Then we create the event handler as an arrow function with one string parameter. We use an `onClick` event property similar to inline DOM events and assign a callback function, `clickHandler`. As soon as the user clicks the page's headline, we display a simple alert box.

Next, we define the component. As you can see, it's a JSX expression that implements the `WeatherProps` interface and returns a JSX element. Inside the component, we use an untagged template literal to create text and add the dynamic weather information with the value from the `weather` attribute, via `props.weather`. Then we return the JSX element and, finally, return and render the weather component, setting `sunny` as the attribute's value.

Paste this code into the `App.tsx` file. The browser should render an `h1` HTML element with the text `The weather is sunny` in the preview pane. When you click the text, an alert box will display it once more. Change the value of the `weather` attribute to display different weather strings.

Writing Class Components

There are two kinds of components in React: class components and function components. The component in Listing 4-2 is a *function component*, which borrows heavily from functional programming. In particular, these components follow the pattern of pure functions: they create some output (JSX elements) based on some input (the `props` argument and the JSX component's attributes). While we emphasize this type of component in this chapter, you should know the basics of class components too.

A *class component* follows the typical patterns of object-oriented programming: it is defined as a class and inherits methods from its parent `React.Component` class. Like all components, it has an argument called `props` and returns a JSX element. Class components also have constructor and `super` functions, and you can use the `this` keyword to refer to the current component's instance.

Of particular value, the internal property `this.state` provides you an interface to store and access information about the component's internal state, such as opened elements, the current image in an image gallery, or, as in the next example, a simple click counter. Of similar importance are the class's *lifecycle* methods, which run during specific lifecycle steps: for example, whenever the component mounts, renders, updates, or unmounts. In Listing 4-3, we use the `componentDidMount` lifecycle method. React runs this method immediately after the component becomes part of the DOM. It is similar to the browser's `DOMContentLoaded` event, with which you might already be familiar.

Listing 4-3 shows the previously created weather component defined as a class component. To practice accessing the component's state, we've added a counter that will count the clicks on the headline element. Because it records the internal component's state, the counter resets on page reload. Paste this code into the `App.tsx` file and click the headline to count up.

```
import React from "react";

export default function App() {
  interface WeatherProps {
    weather: string;
  }
```

```

type WeatherState = {
  count: number;
};

class WeatherComponent extends React.Component<WeatherProps, WeatherState> {
  constructor(props: WeatherProps) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    this.setState({ count: 1 });
  }

  clickHandler(): void {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <h1 onClick={() => this.clickHandler()}>
        The weather is {this.props.weather}, and the counter shows{" "}
        {this.state.count}
      </h1>
    );
  }
}

return (<WeatherComponent weather="sunny" />);
}

```

Listing 4-3: A basic React class component

First we define the custom interface to use for the component's properties. We also define a type to use in the counter we'll create later.

Next, we define the class component, extending the base class `React.Component`. Following object-oriented programming patterns, the constructor calls a `super` function and initializes the component's state. We set our counter to 0. As soon as the browser mounts the component, it calls the lifecycle method `componentDidMount`, changing the component's `count` variable to 1. We modify the click handler to count the number of clicks instead of displaying an alert box, and we call the `render` function. Here we return the JSX elements that display the weather props and the current state as HTML.

Finally, we return the `WeatherComponent`, and React initializes it. The preview pane displays the string `The weather is sunny`, and the counter shows 1. We see from the number 1 that the lifecycle method was indeed called. Each click on the headline increases the number instantly, because of the reactive nature of the component's state. As soon as the state changes, React re-renders the component and updates the view with the current value of the state.

Providing Reusable Behavior with Hooks

Function components can use *hooks* to provide reusable behaviors, such as for accessing a component's state. Hooks are functions that offer simple and reusable interfaces to state and lifecycle features. Listing 4-4 shows the same weather component we created in Listing 4-3, this time written as a function component. It uses hooks instead of lifecycle methods to update the component's counter.

```
import React, { useState,useEffect } from "react";

export default function App() {

  interface WeatherProps {
    weather: string;
  }

  const WeatherComponent = (props: WeatherProps): JSX.Element => {

    const [count, setCount] = useState(0);
    useEffect(() => {setCount(1)},[]);

    return (
      <h1 onClick={() => setCount(count + 1)}>
        The weather is {props.weather},
        and the counter shows {count}
      </h1>
    );
  };

  return (<WeatherComponent weather="sunny" />);
}
```

Listing 4-4: A React function component that uses hooks

We've added two new features to this component: an indicator of the component's state and a way to run code as soon as we mount the component. Therefore, we use the two hooks, `useState` and `useEffect`, by importing them as named imports from the React module, then adding them to the function component. The `useState` hook replaces the `this.state` property from the class component, and the `useEffect` hooks the `componentDidMount` lifecycle method. In addition, we replace the `clickHandler` from the previous example with a simple inline function to update the counter.

Each call to a hook produces an entirely isolated state, so we can use the same hook multiple times in the same component and trust that the state will update. This pattern keeps the hook callbacks small and focused. Also note that the runtime does not hoist hooks. They are called in the order in which we define them in the code.

When you compare Listings 4-3 and 4-4, you should instantly see that the function component is more readable and easier to understand. For this reason, we'll exclusively use function components in the rest of this book.

Working with Built-in Hooks

React provides a collection of built-in hooks. You’ve just seen the most common ones, `useState` and `useEffect`. Another useful hook is `useContext`, for sharing data among components. Other built-in hooks cover more specific use cases to enhance the performance of your application or handle specific edge cases. You can look them up as needed in the React documentation.

You can also create custom hooks whenever you need to break a monolithic component into smaller, reusable packages. Custom hooks follow a specific naming convention. They start with `use`, followed by an action beginning with an uppercase letter. You should define only one functionality per hook to make it easily testable.

This section will guide you through the three most common hooks and the benefits of using them.

Managing the Internal State with `useState`

A pure function uses only the data that is available inside the function. Still, it can react to local state changes, such as the counter in the weather component we created. The `useState` hook is probably the most-used one for handling regional states. This internal component’s state is available only inside the component and is never exposed to the outside.

Because the component state is reactive, React re-renders the component as soon as we update its state, changing the value across the entire component. However, React guarantees that the state is stable and won’t change on re-renders.

The `useState` hook returns the reactive state variable and a setter function used to set the state, as shown in Listing 4-5.

```
const [count, setCount] = useState(0);
```

Listing 4-5: The `useState` hook viewed in isolation

We initialize the `useState` hook with the default value. The hook itself returns the state variable `count` and the setter function we need to modify the state variable’s value, because we cannot modify this variable directly. For example, to set the state variable `count` we created in Listing 4-5 to 1, we need to call the `setCount` function with the new value as a parameter, like this: `setCount(1)`. By convention, the setter function begins with a `set` followed by the state variable’s name.

Handling Side Effects with `useEffect`

Pure functions should rely only on the data passed to them. When a function uses or modifies data outside its local scope, we call this a *side effect*. The simplest example of a side effect is modifying a global variable. This is considered a bad practice both in JavaScript and in functional programming.

Sometimes, however, our components need to interact with the “outside world” or have an external dependency. In these cases, we can use the `useEffect` hook, which handles side effects, providing an escape hatch from the functional aspect of the component. For example, `useEffect` can manage dependencies, call APIs, and fetch data required for the component.

This hook runs after React mounts the component into the layout and the rendering process of the component is completed. It has an optional return object, which runs before the component is unmounted. You can use it for cleanup, for example, to remove event listeners.

One way to use this hook is to observe and react to dependencies. To do this, we can pass it an optional array of dependencies. Any change to one of these dependencies would trigger a rerun of the hook. If the dependency array is empty, the hook won’t depend on any external value and never reruns. This is the case in our weather component, where `useEffect` is executed only after mounting and unmounting the component. It has no external dependencies, so the dependency array remains empty and the hook runs only once.

Sharing Global Data with `useContext` and Context Providers

Ideally, React’s function components would be pure functions that operate only on data passed through the `props` parameter. Alas, a component might sometimes need to consume a shared, global state. In this case, React implements the *context provider* to share global data with a tree of child components.

The context provider wraps the child components, and we can access the shared data with the `useContext` hook. As the context value changes, React automatically re-renders all child components. Thus, it is quite an expensive hook. You shouldn’t use it for datasets that change frequently.

In the full-stack application you’ll build in Part II, you’ll use `useContext` to share session data with child components. Shared contexts are also often employed to keep track of color schemes and themes. Listing 4-6 shows how to consume a theme through a context provider.

```
import React, { useState, createContext, useContext } from "react";

export default function App() {
  const ThemeContext = createContext("");

  const ContextComponent = (): JSX.Element => {

    const [theme, setTheme] = useState("dark");

    return (
      <div>
        <ThemeContext.Provider value={theme}>
          <button onClick={() => setTheme(theme == "dark" ? "light" : "dark")}>
            Toggle theme
          </button>
          <Headline />
        </ThemeContext.Provider>
      </div>
    );
  };
}
```

```

        </ThemeContext.Provider>
      </div>
    );
  };

  const Headline = (): JSX.Element => {
    const theme = useContext(ThemeContext);
    return (<h1 className={theme}>Current theme: {theme}</h1>);
  };

  return (<ContextComponent />);
}

```

Listing 4-6: A complete context provider example

First we import the necessary functions from the React package and use the `createContext` function to initialize the `ThemeContext`. Next, we create the parent component and name it `ContextComponent`. This is the wrapper that holds the context provider and all child components.

In the `ContextComponent`, we create the local theme variable with `useState` and set the stateful variable as the content the context provides. This enables us to change the variable in the context from inside a child component. Because we used a reactive stateful variable for the value, all instances of the theme variable will instantly update across all child components.

We add a button element and toggle the value of the stateful variable between light and dark whenever a user clicks the button. Finally, we create the `Headline` component, which calls the `useContext` hook to get the theme value provided by the `ThemeContext` to all child components. The `Headline` component uses the theme value for the HTML class and displays the current theme.

Exercise 4: Create a Reactive User Interface for the Express.js Server

Let's use your new knowledge and our weather component to create a reactive user interface for the Express.js server. The new React component will allow us to update text on the web page by clicking it.

Adding React to the Server

First we'll include React in our project. For experimentation purposes, you can add the React library and the stand-alone version of the Babel.js transpiler directly inside your HTML head tag. Be aware, however, that this technique is not suitable for production. Transpiling code in the browser is a slow process, and the JavaScript libraries we add here aren't optimized. Using React with a skeleton Express.js server requires a decent number of tedious setup steps and a decent amount of maintenance. We'll use Next.js in Chapter 5 to simplify developing React applications.

Create a folder, named *public*, next to the *package.json* file and then create an empty file called *weather.html* inside it. Add the code in Listing 4-7, which contains our React example with the weather component. Later,

we'll create a new endpoint, */components/weather*, that directly returns the HTML file.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Weather Component</title>
    <script src="https://unpkg.com/react@18/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
  <body>
    <div id="root"></div>

    <script type="text/babel">
      function App() {

        const WeatherComponent = (props) => {

          const [count, setCount] = React.useState(0);
          React.useEffect(() => {
            setCount(1);
          }, []);

          return (
            <h1 onClick={() => setCount(count + 1)}>
              The weather is {props.weather},
              and the counter shows {count}
            </h1>
          );
        };
        return (<WeatherComponent weather="sunny" />);
      }

      const container = document.getElementById("root");
      const root = ReactDOM.createRoot(container);
      root.render(<App />);
    </script>
  </body>
</html>
```

Listing 4-7: The static file */public/weather.html* renders React in the browser.

First we add three React scripts to the *weather.html* file: these are *react.development*, *react.dom.development*, and the stand-alone *babel.js*, which are all similar to the import of React we previously used in the *App.tsx* file. Then we add ReactDOM to let React interact with the DOM. The three files add a global property, React, to *window.object*. We use this property as a global variable to reference React functions. The stand-alone Babel script adds the Babel.js transpiler, which we need to convert the code from JSX to JavaScript.

Next, we add the weather component's code we developed previously. Instead of referencing the *App.tsx* file, we place app functions directly inside

the HTML file and mark the script block as `text/babel`. This type tells Babel to transpile the code inside the script tag into standard JavaScript.

We make a few simple modifications to the weather component's code. First we remove the type annotations, as they are allowed only in TypeScript files. Then, because we are using the browser environment, we prefix the hooks with their global property name, `React`. Finally, we use `ReactDOM` to create the React root container and render the `<App />` component there.

Creating the Endpoint for the Static HTML File

The second file we'll edit is the `index.ts` file in the root directory. We add the highlighted code in Listing 4-8 to add a new entry point, `/components/weather`.

```
import { routeHello, routeAPINames, routeWeather } from "./routes.js";
import express, { Request, Response } from "express";

import path from "path";

const server = express();
const port = 3000;

--snip--
server.get("/components/weather", function (req: Request, res: Response): void {
  const filePath = path.join(process.cwd(), "public", "weather.html");
  res.setHeader("Content-Type", "text/html");
  res.sendFile(filePath);
});

server.listen(port, function (): void {
  console.log("Listening on " + port);
});
```

Listing 4-8: The refactored `index.ts`

To load the static HTML file, import `path` from Node.js's default `path` module. The `path` module provides all kinds of utilities for working with files and directories. In particular, we'll use the `join` function to create a valid path that meets the operation system's format.

We use the default global `process.cwd` function to get the current working directory, and from there, we create the path to our HTML file. Then we add the weather component's entry point and set the response's `Content-Type` header to `text/html`. Finally, we use the `sendFile` function to send to the browser the `weather.html` file we created previously.

Running the Server

We need to transpile the server code to JavaScript, so we run TSC with `npm` on the command line:

```
$ npm run tsc
```

The generated files, *index.js* and *routes.js*, are similar to the previously created ones. TSC doesn't touch the static HTML. The stand-alone Babel.js script converts the JSX code on runtime in the browser. Start the server from your command line:

```
$ node index.js
Listening on 3000
```

Now visit *http://localhost:3000/components/weather-component* in your browser. You see the same text you saw when you rendered the weather component in the React playground, as in Figure 4-2. As soon as you click the text, the click handler increases the reactive state variable, and the counter shows the new value.



Figure 4-2: Browser response from the Node.js web server

You successfully created your first React application. To gain more experience with React, try adding a custom button component for the click counter, with a style attribute that uses a JSX expression to change the background color for odd and even counter values.

Summary

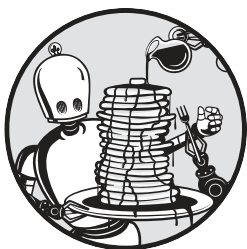
You should now have a solid foundation with which to create your React apps. JSX elements are the building blocks of React components that return JSX to be rendered as HTML in the DOM, via React's virtual DOM. You also explored the difference between class components and modern function components, took a deep dive into React hooks, and used these hooks to build a function component.

If you want to explore React's full potential, take a look at the React tutorials from W3Schools at <https://www.w3schools.com/REACT/DEFAULT.ASP> and those created by the React team at <https://react.dev/learn/tutorial-tic-tac-toe>.

In the next chapter, we'll work with Next.js. Built on top of React, Next.js is a production-ready full-stack web development framework for single-page applications.

5

NEXT.JS



In Chapter 4, you used React to create responsive user interface components. But because React is just a library, building a full-stack application requires additional tools.

In this chapter, we use Next.js, the leading web application framework built on top of React. To create an app with Next.js, you need to know only a few essential concepts. This chapter covers them.

Next.js streamlines the creation of an application's frontend, middleware, and backend. On the frontend, it uses React. It also adds native CSS modules to define styles, and custom Next.js modules to perform routing, image handling, and additional frontend tasks. When it comes to the middleware and the backend, Next.js uses a built-in server to provide the entry points for HTTP requests and a clean API in which to work with request and response objects.

We'll cover its filesystem-based approach to routing, discuss ways to build and render the web pages we deliver to clients, explore adding CSS files to style pages, and refactor our Express.js server to work with Next.js. This chapter uses the traditional *pages* directory to teach you these basic concepts. To learn about Next.js's alternative *app* directory, see Appendix B.

Setting Up Next.js

Next.js is part of the npm ecosystem. While you could manually install all of its required modules by running `npm install next react react-dom` and subsequently create all of your project's files and folders by yourself, there is a much simpler way to set things up: running the `create-next-app` command.

Let's create a sample application to use throughout this chapter. Follow these steps to set up a new empty folder called *sample-next* and build your first Next.js application inside it. Keep the default answers from the setup wizard, and choose to use the traditional *pages* directory instead of the *app* directory:

```
$ mkdir sample-next
$ cd ./sample-next
$ npx create-next-app@latest --typescript --use-npm
--snip--
What is your project named? ... my-app
--snip--
Creating a new Next.js app in /Users/.../my-app.
```

Installing dependencies:

- react
- react-dom
- next

Installing devDependencies:

- eslint
 - eslint-config-next
 - typescript
 - @types/react
 - @types/node
 - @types/react-dom
-

We create a new folder, switch to it, and then initialize a new Next.js project. We use the `npx` command instead of `npm` because, as you learned in Chapter 1, `npx` doesn't require us to install anything as a dependency or development dependency. We mentioned that a typical use case for it is scaffolding, which is precisely what we're doing here.

The `create-react-app` command has a few options, of which only two are relevant to us: the `--typescript` option creates a Next.js project that supports TypeScript, and the `--use-npm` flag selects npm as a package manager.

We accept the default project name, `my-app`, and all the other default settings. The script creates a folder based on the project name containing the

package.json file and a complete sample project with all necessary files and folders. Finally, it installs the dependencies and development dependencies through npm.

NOTE

Instead of setting up a new project, you can use the online playgrounds at <https://codesandbox.io/s/> or <https://stackblitz.com> to run the Next.js code examples from this chapter. Just opt for the pages directory setup instead of app there as well.

Project Structure

Let's explore the boilerplate Next.js app's project structure. Enter the following commands to run it:

```
$ cd my-app
$ npm run dev
```

```
> my-app@0.1.0 dev
> next dev
```

```
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
```

Visit the provided URL in your browser. You should see a default page similar to the one in Figure 5-1 (this welcome page could change depending on your Next.js version).

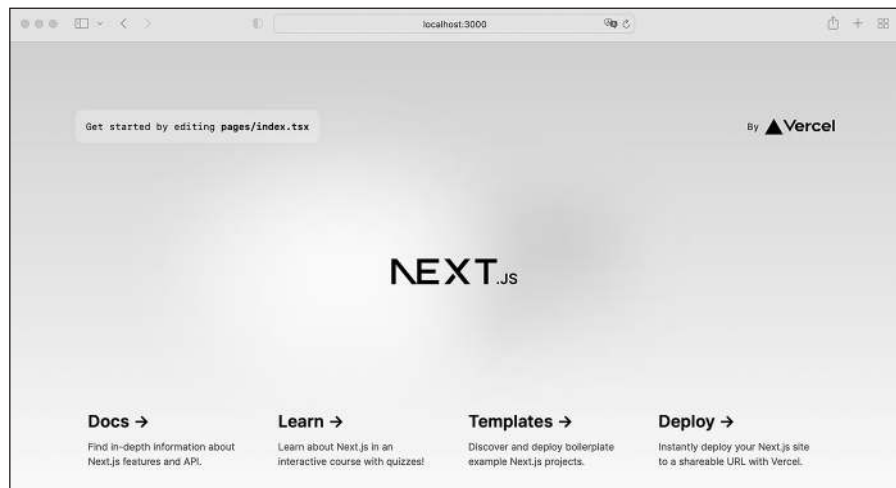


Figure 5-1: The boilerplate Next.js app viewed in a browser

Now open the *my-app* folder that the scaffolding script created, and look around. The *my-app* folder contains a lot of folders, but only three are currently important to you: *public*, *styles*, and *pages*.

The *public* folder holds all static assets, such as custom font files, all images, and files the app makes available for download. We'll link to these assets from the app's HTML and CSS files. The *pages* folder contains all of

the app's routes. Each of its files is an endpoint belonging to a page route or an API route (in the *api* subfolder).

NOTE

Recent versions of Next.js additionally include an `app` directory that you can choose to use for routing as an alternative to the `pages` directory. Because the `app` directory uses more advanced concepts, this chapter covers the simpler `pages` architectural style. However, you can learn more about the `app` directory in Appendix B, where we'll cover its use in detail.

In the *my-app* folder, we also find the *_app.tsx* file, which is Next.js's equivalent to the *App.tsx* file we used in Chapter 4. This is the entry point for the whole application and the place where we'll add our global styles, components, and context providers. Finally, the *styles* folder contains the global CSS files and modules for locally scoped, component-specific files.

Development Scripts

The technologies our app uses, including TypeScript, React, and JSX, don't run directly in the browser. They all require a build pipeline with a reasonably complex transpiler. Next.js provides four command line scripts to simplify development.

The `npx next dev` and `npm run dev` commands starts the application at `http://localhost:3000` in development mode. As a result, Next.js rebuilds and reloads the rendered application in the browser window as soon as we change a file. In addition to this *hot-code* reloading, the development server also displays errors and warning messages to aid the application's development. The installation wizard adds the server to *package.json*'s script section, so we can start it with `npm run dev` as well.

The `npx next build` and `npm run build` commands create the optimized build of our application. They remove unused code and reduce the file size of our scripts, styles, and all other assets. You'll use them for the live deployment. The `npx next start` and `npm run start` commands run the optimized application at `http://localhost:3000` in production mode on the built-in server or in a serverless environment. This production build relies on a previously created build. Hence, we must have first run the `build` command.

Finally, `npx next export` and `npm run export` commands create a standalone version of your application that is independent of the built-in Next.js server and can run on any infrastructure. This version of your app won't be able to use features that require Next.js on the server side, however. Consult the official Next.js documentation at <https://nextjs.org> for a guide to using it.

Routing the Application

When we built our sample Express.js server, we created an explicit routing file that mapped each of the app's endpoints to distinct functions that performed corresponding behavior. Next.js offers a different, perhaps simpler, routing system; it automatically creates the app's routes

based on the files in the *pages* directory. If a file in this folder exports a React component (in the case of a web page) or an async function (in the case of an API), it becomes a valid endpoint, as either an HTML page or an API.

In this section, we'll revisit the routes we created in our Express.js server and remake them using Next.js's routing technique.

Simple Page Routes

For our Express.js server, we manually created a */hello* route in the *index.ts* file. When visited, it returned Hello World! Let's convert this route to a page-based one in Next.js. The simplest kind of page route consists of a file placed directly in the *pages* directory. For example, the *pages/index.tsx* file, created by default, maps to *http://localhost:3000*. To create a simple */hello* route, make a new file, *hello.tsx*, in that directory. Now add to it the code from Listing 5-1.

```
import type { NextPage } from "next";

const Hello: NextPage = () => {
  return (<>Hello World!</>);
}

export default Hello;
```

Listing 5-1: The pages/hello.tsx file

Our Express.js server used the *routeHello* function to return the Hello World! string. Here we need to add a little more code to export a React component. First we import the custom type *NextPage* from the *Next.js* module and use it to create a constant, *Hello*. We assign the constant a fat arrow function that returns a *NextPage*, which is nothing but a custom wrapper for React components. In this case, we return JSX that renders the Hello World! string. Finally, we export the *NextPage* as the file's default export.

Run the server and navigate to *http://localhost:3000/hello*. The page you see should show Hello World! as its content.

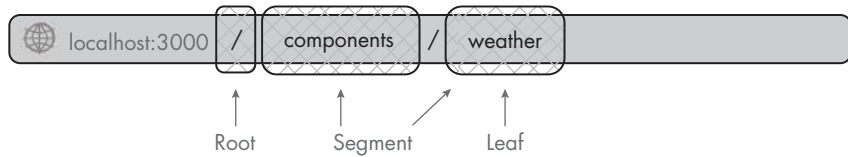
The page looks different from the one in the sample Express.js server. That's because Next.js automatically renders all global styles defined in the *_app.tsx* file to each page. Hence, the font looks different, even though we didn't explicitly define any styles in the *hello.tsx* file.

Nested Page Routes

Nested routes, such as */components/weather* from the sample Express.js server, are logical subroutes of other routes. In other words, *weather* is nested inside the *components* entry point. You've probably already guessed how we create a nested route with Next.js's page-routing pattern. Yes, we merely create a subfolder, and Next.js maps the folder structure to the URL schema.

Create a new folder, *components*, inside the *pages* folder and add a new file, *weather.tsx*, there. Figure 5-2 depicts the relationship between the URL *components/weather* and the file structure *pages/components/weather.tsx*.

Browser URL:



Files:

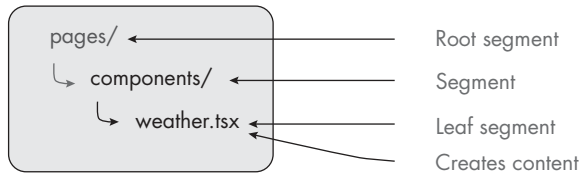


Figure 5-2: The relationship between the URL *components/weather* and the file structure *pages/components/weather.tsx*

Our *pages* folder is the root of the URL, and each nested folder becomes a URL segment. The file that exports the *NextPage* is the *leaf segment*, or the final part of the URL. For this file, we reuse the *weather* component code we wrote in Chapter 4, shown in Listing 5-2.

```
import type { NextPage } from "next";
import React, { useState, useEffect } from "react";

const PageComponentWeather: NextPage = () => {

  interface WeatherProps {
    weather: string;
  }

  const WeatherComponent = (props: WeatherProps) => {

    const [count, setCount] = useState(0);
    useEffect(() => {
      setCount(1);
    }, []);

    return (
      <h1 onClick={() => setCount(count + 1)}>
        The weather is {props.weather},
        and the counter shows {count}
      </h1>
    );
  };
};
```

```
    return (<WeatherComponent weather="sunny" />);  
  };  
  
export default PageComponentWeather;
```

Listing 5-2: The pages/components/weather.tsx file

The only difference from the functional component created in Chapter 4 is that we wrap the code in a function that returns a `NextPage`, which we then export as the default export. This is consistent with the page we created in Listing 5-1 and follows Next.js's pattern requirement.

Visit the new page at <http://localhost:3000/components/weather> in the browser. It should look similar to Figure 5-3.



Figure 5-3: The pages/components/weather.tsx file is rendered at the /components/weather URL in the browser.

You should recognize the click-handler functionality you saw in Chapter 4.

API Routes

In addition to a user-friendly interface, a full-stack application might also need a machine-readable interface. For example, the Food Finder application you'll create in Part II will provide an API to external services so that a mobile app or a third-party widget can display our wish list. As JavaScript-driven full-stack developers, the most common API formats we'll use are GraphQL and REST, and we talk about these in depth in Chapter 6. Here we will create REST APIs, which we like for their simplicity.

With Next.js, we can design and create APIs via the same patterns we use for pages. Each file in the `pages/api/` folder is a single API endpoint, and we can define nested API routes in the same way we define nested page routes. However, unlike page routes, API routes are not React components. Instead, they are async functions that take two parameters, `NextApiRequest` and `NextApiResponse`, and return a `NextApiResponse` and JSON data.

There are two caveats you need to remember when it comes to API routes. First, they do not specify a *Cross-Origin Resource Sharing (CORS)* header by default. This set of HTTP headers, most notably the `Access-Control-Allow-Origin` header, lets a server define the origins from which client-side scripts can request resources. If you want client-side scripts running in websites on third-party domains to access your API endpoints,

you'll need to add additional middleware to enable CORS directly in the Next.js server. Otherwise, external requests will prompt a CORS error. This isn't specific to Next.js; Express.js and most other server frameworks require you to do the same.

The second caveat is that static exports done by running `next export` do not support API routes. They rely on the built-in Next.js server and cannot run as static files.

We used one API route, *api/names*, in the Express.js server. Now let's refactor the code and convert it to a Next.js API route. As before, create a new file, *names.ts*, and place it in the *api* folder. Because API routes return an async function instead of JSX, we use the *.ts* extension, not the *.tsx* extension used for JSX code. Paste the code from Listing 5-3 into the file.

```
import type { NextApiRequest, NextApiResponse } from "next";

type responseItemType = {
  id: string;
  name: string;
};

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
): Promise<NextApiResponse<responseItemType[]> | void> {
  const url = "https://www.usemodernfullstack.dev/api/v1/users";
  let data;
  try {
    const response = await fetch(url);
    data = (await response.json()) as responseItemType[];
  } catch (err) {
    return res.status(500);
  }
  const names = data.map((item) => {
    return { id: item.id, name: item.name };
  });
  return res.status(200).json(names);
}
```

Listing 5-3: The pages/api/names.ts file

First we import the custom types for the API request and response from the Next.js package. Then we define the custom type for the API response. In Chapter 3, we created the same type for typing the `await` call in the *routes.ts* file. We're using the same code and `await` call here, so we've reused the type as well. We then create and directly export the API handler function mentioned earlier. You learned in Chapter 2 that async functions need to return a promise as their return type. Therefore, we wrap this API response in a promise.

The code in the function's body is similar to the code in the *routeAPINames* function from Chapter 4. It makes an API request to fetch the user data,

converts the received data into the desired return format, and finally returns the data. However, we need to make a few modifications. First, instead of returning an error string, we return an API response with no content and a generic status code of *500*, for an *Internal Server Error*.

The second adjustment involves the data mapping. Previously, we returned a string that rendered in the browser. Now, instead of this string, we return a JSON object. Therefore, we modify the `array.map` function to create an array of objects. Finally, we change the return statement to return the API response with the names object as JSON and a status code of *200: OK*.

Now open the new API route in the browser at `http://localhost:3000/api/names`. You should see the API response shown in Figure 5-4.

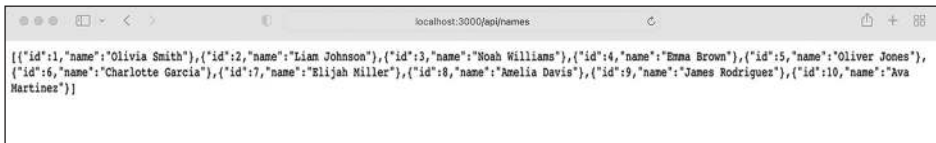


Figure 5-4: The `pages/api/names.ts` file rendered from `/api/names` in the browser

Dynamic URLs

You now know how to create page and API routes, which are the foundation of any full-stack application. However, you might be wondering how to create *dynamic* URLs, which change based on input. We often use dynamic URLs for profile pages, where the user's name becomes part of the URL. In fact, we implemented a dynamic URL in the Express.js server's weather API when we defined the route `/api/weather/:zipcode` in the `index.ts` file. There, `zipcode` was a dynamic parameter, or a dynamic leaf segment, whose value was provided by the `req.params.zipcode` function.

Next.js uses a slightly different pattern for dynamic URLs. Because it creates the routes based on folders and files, we need to define dynamic segments through their filenames by wrapping the variable portion in square brackets `[]`. The dynamic route `/api/weather/:zipcode` from the Express.js server would thus translate to the file `/api/weather/[zipcode].ts`.

Let's create a dynamic route in our sample Next.js application that mimics the `/api/weather/:zipcode` route from the Express.js server. Make a new folder, `weather`, in the `api` folder, and place a file named `[zipcode].ts` in it. Then paste the code from Listing 5-4 into the file.

```
import type { NextApiRequest, NextApiResponse } from "next";

type WeatherDetailType = {
  zipcode: string;
  weather: string;
  temp?: number;
};
```

```
export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
): Promise<NextApiResponse<WeatherDetailType> | void> {

  return res.status(200).json({
    zipcode: req.query.zipcode,
    weather: "sunny",
    temp: 35
  });
}
```

Listing 5-4: The api/weather/[zipcode].ts file

This code should be familiar to you, as it follows the basic outline of an API route in Next.js. We import the necessary types, then define a custom type, `WeatherDetailType`, and use it as the type for the data returned by the function. (By the way, this is the same type definition we created in Chapter 3.) In the function's body, we return the response with a status code of `200: OK` and a JSON object. We fill the `zipcode` property with the ZIP code from the dynamic URL parameter, retrieved with `req.query.zipcode`.

When you run the server, the browser should show the JSON response with the dynamic URL parameter in the response type. If you navigate to `http://localhost:3000/api/weather/12345`, you should see the API response. If you change the “12345” part of the URL and request the data again, the response data should change accordingly.

Note that the dynamic route `/api/weather/[zipcode].ts` matches `/api/weather/12345` and `/api/weather/54321` but not sub-paths of those routes, such as `/api/weather/location/12345` or `/api/weather/location/54321`. For this, you'll need to use a *catch all* API route, which includes all paths that are inside the current path. You can create a catch all route by adding three dots (...) in front of the filename. For example, the catch all route `/api/weather/[...zipcode].ts` could handle all four API endpoints mentioned in this paragraph.

Styling the Application

To add styles to our Next.js application, we create regular CSS files, written without the vendor prefixes used in other frameworks. Later, Next.js's postprocessor will add necessary properties to generate backward-compatible styles. While CSS syntax is beyond the scope of this book, this section describes how to use Next.js's two kinds of CSS styles: global styles and locally scoped component styles, defined in CSS modules.

Global Styles

Global styles affect all pages of an app. We stumbled across this behavior when we rendered the *hello.tsx* file; the page used CSS even though we hadn't added any style information ourselves.

Practically speaking, global styles are just regular CSS files. They aren't modified during the build, and their class names are guaranteed to stay the same. Therefore, we can use them as regular CSS classes across the application. We import these CSS files in the app's entry point, the *pages/_app.tsx* file. Take a look at those in the boilerplate project. You should see a line of code similar to Listing 5-5.

```
import "@styles/globals.css";
```

*Listing 5-5: Importing global styles in the *_app.tsx* file*

Of course, you can adjust the filename and location of the imported styles or import multiple files. Try playing around by adding a few styles in the *global.css* file and some regular CSS classes to the HTML elements in the *hello.tsx* file. Then visit the page at <http://localhost:3000/hello> to see how it changed.

Component Styles

In Chapter 4, you saw that React lets us create user interfaces out of independent, reusable components. Global styles aren't the best approach for styling independent components, as they require us to keep track of the names we've already used in various components, and if we import components from a previous project, we risk having the CSS classes collide with one another.

We need the CSS classes to be scoped to individual modules to work efficiently with modularized components. There are multiple architectural patterns for implementing this. For example, using the Block Element Modifier methodology, you can manually scope the styles to a component or a user interface block.

Luckily, we don't need to bother with such a clumsy solution. Next.js lets us use CSS modules that are scoped during the build process. These CSS modules follow the naming convention *<component>.module.css*. The compiler automatically prefixes each CSS class name inside the module with the component's name and a unique identifier. This enables you to use the same style names for multiple components without issue.

The actual CSS you write won't have these prefixes. For example, look at the *Home.module.css* file inside the *styles* folder, shown in Listing 5-6.

```
.container {  
  padding: 0 2rem;  
}
```

*Listing 5-6: Regular CSS code in *styles/Home.module.css**

One problem is that, because the build process modifies the class names and prefixes them, we can't directly use these styles in our other files. Instead, we must import the styles and treat them like a JavaScript object. Then we can refer to them as a property of the styles object. For example, the *pages/index.tsx* file in Listing 5-7 uses the container class from Listing 5-6, providing an example of how to use scoped styles.

```
import styles from "../styles/Home.module.css"
--snip--
const Home: NextPage = () => {
  return (
    <div className={styles.container}>
      --snip--
    </div>
  );
};
```

Listing 5-7: Using styles from the CSS module styles/Home.module.css in the index.tsx file

This code imports the CSS file into a constant called `styles`. Now all the CSS class names will be available as properties of the styles object. In JSX, we use variables wrapped in curly brackets (`{}`), so we add a reference to the container class as `{styles.container}`.

You can now build APIs and custom-styled pages out of React components. The next section introduces useful custom components that Next.js provides to enhance your full-stack application.

Built-in Next.js Components

Next.js provides a set of custom components. Each of these addresses one specific use case: for example, accessing internal page properties such as the page title or SEO metadata (`next/head`), improving the app's overall rendering performance and user experience (`next/image`), or enabling the application's routing (`next/link`). We'll use the Next.js components covered in this chapter in the full-stack application in Part II, where you can see them applied in practice. For additional attributes and niche use cases, refer to the Next.js documentation.

The next/head Component

The `next/head` component exports a custom Next.js-specific `Head` component. We use it to set a page's HTML title and meta elements, which are found inside an HTML head component. To improve SEO ranking and enhance usability, each page should have its own metadata. Listing 5-8 shows an example of the *hello.tsx* page from Listing 5-1 with a customized title and meta element.

It is important to remember that the `Head` elements are not merged across pages. Next.js's client-side routing removes the content of the `Head` element during the page transition.

```
import type { NextPage } from "next";
import Head from "next/head";

const Hello: NextPage = () => {
  return (
    <div>
      <Head>
        <title>Hello World Page Title</title>
        <meta property="og:title" content="Hello World" key="title" />
      </Head>
      <div>Hello World!</div>
    </div>
  );
};

export default Hello;
```

Listing 5-8: The pages/hello.tsx file with a customized title and meta element

We import the Head element from the next/head component and add it to the returned JSX element, placing it above the existing content and wrapping both in another div element because we need to return one element instead of two.

The next/link Component

The next/link component exports a Link component. This component is built on top of the React Link element. We use it instead of an HTML anchor tag when we want to link to another page in the application, enabling client-side transitions between pages. When clicked, the Link component updates the browser DOM with the new DOM, scrolls to the top of the new page, and adjusts the browser history. Furthermore, it provides built-in performance optimizations, prefetching the linked page and its data as soon as the Link component enters the *viewport* (the currently visible part of the website). This background prefetch enables smooth page transitions. Listing 5-9 adds the Next.js Link element to the page from the previous listing.

```
import type { NextPage } from "next";
import Head from "next/head";
import Link from "next/link";

const Hello: NextPage = () => {
  return (
    <div>
      <Head>
        <title>Hello World Page Title</title>
        <meta property="og:title" content="Hello World" key="title" />
      </Head>
      <div>Hello World!</div>
      <div>
        Use the HTML anchor for an
```

```

        <a href="https://nostarch.com" > external link</a>
        and the Link component for an
        <Link href="/components/weather"> internal page
        </Link>
        .
      </div>
    </div>
  );
};

export default Hello;

```

Listing 5-9: The pages/hello.tsx file with an external link and an internal next/link element

We import the component, then add it to the returned JSX element. For comparison purposes, we use a regular HTML anchor to link to the No Starch Press home page and the custom Link to connect to the weather component page in our Next.js application. In the app, try clicking both links to see the difference.

The next/image Component

The next/image component exports an Image component used to display images. This component is built on top of the native HTML element. It handles common layout requirements, such as filling all available space and scaling images. The component can load modern image formats, such as AVIF and WebP, and serve the image with the correct size for the client's screen. Furthermore, you have the option to use blurred placeholder images and lazy-load the actual image as soon as it enters the viewport; this enforces the visual stability of your website by preventing *cumulative layout shifts*, which occur when an image renders after the page, causing the page content to shift down. Cumulative layout shifts are considered a bad user experience, and they can make the user lose their focus. Listing 5-10 provides a basic example of the next/image component.

```

import type { NextPage } from "next";
import Head from "next/head";
import Link from "next/link";
import Image from "next/image";

const Hello: NextPage = () => {
  return (
    <div>
      <Head>
        <title>Hello World Page Title</title>
        <meta property="og:title" content="Hello World" key="title" />
      </Head>
      <div>Hello World!</div>
      <div>
        Use the HTML anchor for an <a href="https://nostarch.com">
        external link</a> and the Link component for an
        <Link href="/components/weather"> internal page</Link>.
      </div>
    </div>
  );
};

```

```

        <Image
          src="/vercel.svg"
          alt="Vercel Logo"
          width={72}
          height={16}
        />
      </div>
    </div>
  );
};
export default Hello;

```

Listing 5-10: The pages/hello.tsx file using the next/image element

Here we display the Vercel logo from our application's *public* folder. First we import the component from the *next/image* package. Then we add it to the page content. The syntax and the properties of our minimal example are similar to the HTML *img* element. You can read more about the component's advanced properties in the official documentation at <https://nextjs.org/docs/api-reference/next/image>.

Pre-rendering and Publishing

While you can start building full-stack Next.js applications with the information you've learned so far, you'll find it useful to know one more advanced topic: the different ways to render and publish your application and their implications for its performance.

Next.js provides three options for pre-rendering your app with its built-in server. The first, *static site generation (SSG)*, generates the HTML at build time. Thus, each request will always return the same HTML, which remains static and is never re-created. The second option, *server-side rendering (SSR)*, generates new HTML files on each request, and the third, *incremental static regeneration (ISR)*, combines both approaches.

Next.js lets us choose our pre-rendering option on a per-page basis, meaning the full-stack application can contain pages with SSG, SSR, and ISR, as well as client-side rendering for some React components. You can also create a complete static export of your site by running `next export`. The exported application will run independently on all infrastructures, as it doesn't need the built-in Next.js server.

To gain experience with these rendering approaches, we'll create a new page that displays the data from our names API for each rendering option. Create a new folder, *utils*, next to the *pages* folder and add an empty file, *fetch-names.ts*, to it. Then add the code in Listing 5-11. This utility function calls the remote API and returns the dataset.

```

type responseItemType = {
  id: string;
  name: string;
};

```

```

export const fetchNames = async () => {
  const url = "https://www.usemodernfullstack.dev/api/v1/users";
  let data: responseItemType[] | [] = [];
  let names: responseItemType[] | [];
  try {
    const response = await fetch(url);
    data = (await response.json()) as responseItemType[];
  } catch (err) {
    names = [];
  }
  names = data.map((item) => { return { id: item.id, name: item.name } });

  return names;
};

```

Listing 5-11: The async utility in utils/fetch-names.ts

After defining a custom type, we create a function and directly export it. This function contains the code from the previously created *names.ts* file, with two adjustments: first we need to define the data array as possibly empty; next, we return an empty array instead of an error string if the API call fails. This change means that we don't need to verify the type before iterating over the array when we generate the JSX string.

Server-Side Rendering

Using SSR, Next.js's built-in Node.js server creates an application's HTML in response to each request. You should use this technique if your page depends on fresh data from an external API. Unfortunately, SSR is slower in production, because the pages aren't easily cacheable.

To use SSR for a page, export an additional async function, *getServerSideProps*, from that page. Next.js calls this function on every request and passes the fetched data to the page's props argument to pre-render it before sending it to the client.

Try this out by creating a new file, *names-ssr.tsx*, in the *pages* folder. Paste the code from Listing 5-12 into the file.

```

import type {
  GetServerSideProps,
  GetServerSidePropsContext,
  InferGetServerSidePropsType,
  NextPage,
  PreviewData
} from "next";
import { ParsedUrlQuery } from "querystring";
import { fetchNames } from "../utils/fetch-names";

type responseItemType = {
  id: string;
  name: string;
};

```

```

const NamesSSR: NextPage = (props: InferGetServerSidePropsType<typeof getServerSideProps>) => {

  const output = props.names.map((item: responseItemType, idx: number) => {
    return (
      <li key={`name-${idx}`}>
        {item.id} : {item.name}
      </li>
    );
  });

  return (
    <ul>
      {output}
    </ul>
  );
};

export const getServerSideProps: GetServerSideProps = async (
  context: GetServerSidePropsContext<ParsedUrlQuery, PreviewData>
) => {

  let names: responseItemType[] | [] = [];
  try {
    names = await fetchNames();
  } catch(err) {}
  return {
    props: {
      names
    }
  };
};

export default NamesSSR;

```

Listing 5-12: A basic page that displays data with SSR, page/names-ssr.tsx

To use Next.js's SSR, we export the additional async function, `getServerSideProps`. We also import necessary functionality from the *next* and *querystring* packages and the `fetchNames` function we created earlier. Then we define the custom type for the response to the API request. It's the same custom type we used in Chapter 3.

Next, we create the page and store the export as the default one. The page returns a `NextPage` and takes the default properties for this page type. We iterate over the `props` parameter's `names` array and create a JSX string that we render and return to the browser. Then we define the `getServerSideProps` function, which gets the data from the API. We return the created dataset from the async function and pass it to the `NextPage` inside the page properties.

Navigate to the new page at <http://localhost:3000/names-ssr>. You should see the list of the usernames.

Static Site Generation

SSG creates the HTML files only once and reuses them for every request. It is the recommended option, because pre-rendered pages are easy to cache and fast to deliver. For example, a content delivery network can easily pick up your static files.

Usually, SSG applications have a lower *time to first paint*, or the time it takes after a user requests the page (by, for example, clicking a link) until the content appears in the browser. SSG also reduces *blocking time*, or the time it takes until the user can actually interact with the page's content. Good scores in these metrics indicate a responsive website, and they are part of Google's scoring algorithm. Hence, these pages have increased SEO rankings.

If your page relies on external data, you can still use SSG by exporting an additional async function, `getStaticProps`, from the page's file. `Next.js` calls this function at build time, passes the fetched data to the page's props argument, and pre-renders the page with SSG. Of course, this works only if the external data isn't dynamic.

Try creating the same page as in the SSR example, this time with SSG. Add a new file, *names-ssg.tsx*, in the *pages* folder and then paste in the code shown in Listing 5-13.

```
import type {
  GetStaticProps,
  GetStaticPropsContext,
  InferGetStaticPropsType,
  NextPage,
  PreviewData,
} from "next";
import { ParsedUrlQuery } from "querystring";
import { fetchNames } from "../utils/fetch-names";

type responseItemType = {
  id: string,
  name: string,
};

const NamesSSG: NextPage = (props: InferGetStaticPropsType<typeof getStaticProps>) => {

  const output = props.names.map((item: responseItemType, idx: number) => {
    return (
      <li key={`name-${idx}`}>
        {item.id} : {item.name}
      </li>
    );
  });

  return (
    <ul>
      {output}
    </ul>
  );
};
```



```

export const getStaticProps: GetStaticProps = async (
  context: GetStaticPropsContext<ParsedUrlQuery, PreviewData>
) => {

  let names: responseItemType[] | [] = [];
  try {
    names = await fetchNames();
  } catch (err) {}

  return {
    props: {
      names
    }
  };
};

export default NamesSSG;

```

Listing 5-13: A page that displays data with SSG, page/names-ssg.tsx

The code is mostly identical to Listing 5-9. We just need to change the SSR-specific code to use SSG. Therefore, we export `getStaticProps` instead of `getServerSideProps` and adjust the types accordingly.

When you visit the page, it should look similar to the SSR page. But instead of requesting fresh data on each visit to `http://localhost:3000/names-ssg`, the data is requested only once, on page build.

Incremental Static Regeneration

ISR is a hybrid of SSG and SSR that runs purely on the server side. It generates the HTML on the server during the initial build and sends this pre-generated HTML the first time a page is requested. After a specified time has passed, Next.js will fetch the data and regenerate the page on the server in the background. In the process, it invalidates the internal server cache and updates it with the new page. Every subsequent request will receive the up-to-date page. Like SSG, ISR is less costly than SSR and increases a page's SEO ranking.

To enable ISR in SSG pages, we need to add a property to revalidate `getStaticProp`'s return object. We define the validity of the data in seconds, as shown in Listing 5-14.

```

return {
  props: {
    names,
    revalidate: 30
  }
};

```

Listing 5-14: Changing `getServerSideProps` to enable ISR

We add the `revalidate` property with a value of 30. As a result, the custom Next.js server will invalidate the current HTML 30 seconds after the first page request.

Client-Side Rendering

A completely different approach, *client-side rendering* involves first generating the HTML with SSR or SSG and sending it to the client. The client then fetches additional data at runtime and renders it in the browser DOM. Client-side rendering is a good choice when working with highly flexible, constantly changing datasets, such as real-time stock market or currency prices. Other sites use it to send a skeleton version of the page to the client and later enhance it with more content. However, client-side rendering lowers your SEO performance, as its data can't be indexed.

Listing 5-15 shows the page we created earlier, configured for client-side rendering. Create a new file, *names-csr.tsx*, in the *pages* folder and then add the code to it.

```
import type {
  NextPage
} from "next";
import { useEffect, useState } from "react";
import { fetchNames } from "../utils/fetch-names";

type responseItemType = {
  id: string,
  name: string,
};

const NamesCSR: NextPage = () => {
  const [data, setData] = useState<responseItemType[] | []>();
  useEffect(() => {
    const fetchData = async () => {
      let names;
      try {
        names = await fetchNames();
      } catch (err) {
        console.log("ERR", err);
      }
      setData(names);
    };
    fetchData();
  });

  const output = data?.map((item: responseItemType, idx: number) => {
    return (
      <li key={`name-${idx}`}>
        {item.id} : {item.name}
      </li>
    );
  });

  return (
    <ul>
      {output}
    </ul>
  );
};
```

```
};  
  
export default NamesCSR;
```

Listing 5-15: The client-side rendered page, `page/names-csr.tsx`

This code differs significantly from the previous examples. Here we import the `useState` and `useEffect` hooks. The latter one will fetch the data after the page is already available. As soon as the `fetchNames` function returns the data, we use the `useState` hook and the reactive data state variable to update the browser DOM.

We cannot declare the `useEffect` hook as an `async` function, because it returns either an undefined value or a function, whereas an `async` function returns a promise, and therefore TSC would throw an error. To avoid this, we need to wrap the `await` call in an `async` function, `fetchData`, and then call that function.

The page configured for client-side rendering should look similar to the other versions. But when you visit `http://localhost:3000/names-csr`, you might see a white flash. This is the page waiting for the asynchronous API request.

To get a better feel for the different rendering types, modify the code for each example in this section to use the API `https://www.usemodernfullstack.dev/api/v1/now`, which returns an object with the timestamp of the request.

Static HTML Exporting

The next `export` command generates a static HTML version of your web application. This version is independent of the built-in Node.js-based Next.js web server and can run on any infrastructure, such as an Apache, NGINX, or IIS server.

To use this command, your page must implement `getStaticProps`, as in SSG. This command won't support the `getServerSideProps` function, ISR, or API routes.

Exercise 5: Refactor Express.js and React to Next.js

Let's refactor the React and Express.js applications from the previous chapters into a Next.js application that we'll expand in the upcoming chapters. As a first step, we'll summarize the functionality we need to build. Our application has an API route, `api/names`, that returns usernames, and another API route, `api/weather/:zipcode`, that returns a static JSON object and the URL parameter. We used it to understand dynamic URLs. In addition, we created pages at `/hello` and `component/weather`.

Throughout this chapter, we've already refactored these various elements to work with Next.js's routing style. In this exercise, we'll put it all together. Follow the steps in "Setting Up Next.js" on page 70 to initialize the Next.js application. Within the `sample-next` folder, name your application `refactored-app`.

Storing Custom Interfaces and Types

We create a new file, *custom.d.ts*, in the root of the project to store our custom interface and type definitions (Listing 5-16). It is similar to the one we used in Chapters 3 and 4. The main difference is that we add custom types for the Next.js application.

```
interface WeatherProps {  
  weather: string;  
}  
  
type WeatherDetailType = {  
  zipcode: string;  
  weather: string;  
  temp?: number;  
};  
  
type responseItemType = {  
  id: string;  
  name: string;  
};
```

Listing 5-16: The *custom.d.ts* file

We'll use the custom interface *WeatherProps* for the *props* argument of the page that displays the weather components, *components/weather*. The *WeatherDetailType* is for the API route *api/weather/:zipcode*, which uses a dynamically fetched ZIP code. Finally, we use *responseItemType* in the API route *api/names* to type the fetch response.

Creating the API Routes

Next, we re-create the two API routes from the Express.js server. Earlier sections of this chapter showed this refactored code. For the *api/names* route, create a new file, *names.ts*, in the *api* folder, then add the code from Listing 5-3. Refer to that section for a detailed explanation of the code.

Migrate the dynamic route *api/weather/:zipcode* from the Express.js server to the Next.js application by creating a *[zipcode].js* file in the *api* folder and adding the code from Listing 5-4, shown in “Dynamic URLs” on page 77. You can refer to that section for more details.

Creating the Page Routes

Now we work on the pages. First, for the simple page *hello.tsx*, we create a new file in the *pages* folder and add the code from Listing 5-10. This code renders the Hello World! example and uses the custom Next.js components *Head*, *Link*, and *Image*, all of which are explained in detail in “Built-in Next.js Components” on page 80.

The second page is the nested route *pages/components/weather.tsx*. As before, we create a new file, *weather.tsx*, in a folder called *components*, within the *pages* folder. Add the code from Listing 5-2. This listing uses the *useState*

and `useEffect` hooks to create a reactive user interface. We can remove the custom interface definition for the `WeatherProps` from this file. The `custom.d.ts` file already adds them to TSC.

Running the Application

Start the application with the `npm run dev` command. Now you can visit the same routes we created for the Express.js server and see that they are functionally the same. Congratulations! You created your first Next.js-based full-stack application. Play around with the code and try using global and component CSS to style your pages.

Summary

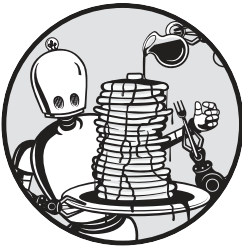
Next.js adds the missing functionality needed to create full-stack applications with React. After scaffolding a sample project and exploring the default file structure, you learned how to create page and API routes in the framework. You also learned about global- and component-level CSS, Next.js's four built-in command line scripts, and its most useful custom components.

We also discussed the different ways to render content and pages with Next.js and when to choose each option. Finally, you used the code from this chapter to quickly migrate the Express.js application you built in the previous chapters to Next.js. To continue your adventures in this useful framework, I recommend the official tutorials at <https://nextjs.org>.

In the next chapter, we'll explore two types of web APIs: the standard RESTful APIs and modern GraphQL.

6

REST AND GRAPHQL APIS



An *API* is a generic pattern used to connect computers or computer programs. Unlike a user interface, it's designed to be accessed not by a user but by another piece of software.

One purpose of APIs is to hide the internal details of a system's workings while exposing a standardized gateway to the system's data or functionality.

As a full-stack developer, you'll usually interact with, or *consume*, two kinds of APIs: internal and third-party. When querying an internal API, you're consuming data from your own systems, typically from your own database or service. Private APIs are not available to outside parties. For example, your bank might use private APIs to check your credit score or account balance on its internal systems and display them in your online banking profile.

Third-party APIs provide access to data from an external system. For example, the OAuth login you'll implement in Part II uses an API. You

might also use an API to fetch a social media feed or weather information from an external provider to display on your website. Because external APIs are exposed to the public, you can reach them through a public URL, and they document the conventions you should use to access their data in an *API contract*. This contract defines the format of the communications, the parameters the API expects, and the possible responses you might receive for each request. We briefly discussed API and function contracts and why you should type them in Chapter 3.

Full-stack web development primarily uses two types of APIs, REST and GraphQL, both of which transmit data over HTTP. This chapter covers these.

REST APIs

REST is an architectural pattern used to design RESTful web APIs. These APIs are essentially a set of URLs, each of which provides access to a single resource. They rely on the use of HTTP methods and the standard HTTP status codes to transmit data and accept URL-encoded or request header parameters. Typically, they respond with the requested data in JSON or plaintext.

In fact, you've already built your first REST API. Recall the Next.js server you created in Exercise 5 on page 89, which provided the `api/weather/:zipcode` endpoint. So far, we've used this endpoint to play with Next.js's routing, understand dynamic URLs, and learn how to access query parameters. You'll soon see, however, that this API follows REST conventions: to access it, we used the HTTP GET method to consume the URL endpoint and received a JSON response with an HTTP status code of `200: OK`. Common status code ranges are `2XX` for successful requests and `3XX` for redirects. If the request fails, we see the `4XX` range to indicate a client-related error, such as `401: Unauthorized`, and `5XX` for server errors, often the generic `500: Internal Server Error`.

As full-stack developers, we might sometimes create our own APIs; more often, though, we'll find ourselves consuming third-party APIs. Why might we consume, say, a third-party weather API? Well, imagine that we want our app to display the current weather at multiple remote locations. Instead of setting up and maintaining various weather stations on our own and then reading the data from the sensors, which would involve both providing and consuming an API for each of them, we could consume data from a third-party API offered by an existing weather service. Our code might call that API, pass in a ZIP code as a parameter, and receive the weather data for this location in a predetermined format. We'd then display this data on our website.

RESTful APIs enable us to interact with data without knowing anything about how that data was stored or what underlying technology provided it. If you follow an API's specifications, you should receive the requested data,

even if the underlying technology or architecture changes. Beyond this, there are a handful of requirements for an API to be considered RESTful.

The URL

A unique URL provides an interface to a RESTful API. Each of a provider's APIs typically has the same base URL, called the *root entry point*, such as `http://localhost:3000/api`. You can think of this as the APIs' family name. Often, you'll see a version number added to the root entry point, because a provider might have multiple versions of an API. For example, there might be the legacy `http://localhost:3000/api/v1` and an updated `http://localhost:3000/api/v2`. To honor this pattern, you can create a folder *v1* inside the *api* folder and move the REST API code there.

NOTE

Other common ways of versioning an API include custom headers and query strings. In the first case, the client would request the API with a custom `Accept-Version` header and receive a matching `Content-Version` header. In the second case, an API request would use `?version=1.0.0` as a query parameter in the URL.

The next part of the API's URL is the path, often called the *endpoint*. It specifies the resource we want to query (for example, the weather API). API specifications usually mention only the endpoint itself, such as `/v1/weather`, leaving the root entry point implied. The URL generally also accepts parameters. These can be path parameters that are part of the URL, like in our ZIP code API endpoint, `/v1/weather/{zipcode}`, or they can be query parameters, which are added as encoded key-value pairs after an initial question mark, as in `/v1/weather?zipcode=<zipcode>`. By convention, path parameters are usually used to refer to a resource or resources, and query parameters are used to perform operations on the data returned, like sorting or filtering.

The Specification

The resources themselves are separate from the representations returned to the client. In other words, the server might send data in formats like HTML, XML, JSON, or others, regardless of the way in which the data is stored in the application's database. You can learn about an API's response format in its *specification*, which serves as the manual for an API. One excellent way to document your APIs is with the OpenAPI format, which is widely used in the industry and is part of the Linux Foundation. You can use the Swagger graphical editor at <https://editor.swagger.io> to experiment with it.

For example, Listing 6-1 shows a specification for the `/v1/weather/{zipcode}` endpoint, written as JSON. Paste the code into the Swagger editor to explore the generated documentation in a more user-friendly manner.

```

{
  "openapi": "3.0.0",
  "info": {
    "title": "Sample Next.js - OpenAPI 3.x",
    "description": "The example APIs from our Next.js application",
    "version": "1.0.0"
  },
  "servers": [
    { "url": "https://www.usemodernfullstack.dev/api/" },
    { "url": "http://localhost:3000/api/" }
  ],
  "paths": {
    "/v1/weather/{zipcode}": {
      "get": {
        "summary": "Get weather by zip code",
        "parameters": [
          {
            "name": "zipcode",
            "in": "path",
            "description": "The zip code for the location as string.",
            "required": true,
            "schema": {
              "type": "string",
              "example": 96815
            }
          }
        ],
        "responses": {
          "200": {
            "description": "Successful operation",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/weatherDetailType"
                }
              }
            }
          }
        }
      }
    }
  },
  "components": {
    "schemas": {
      "weatherDetailType": {
        "type": "object",
        "properties": {
          "zipcode": {
            "type": "string",
            "example": 96815
          },
          "weather": {
            "type": "string",
            "example": "sunny"
          }
        }
      }
    }
  }
}

```


requests of unauthenticated users, those APIs typically provide a token that users should include in subsequent requests. Consumers send this token as part of the request's data or in the HTTP Authorization header. We'll provide more details about authorization tokens and how they work in Chapter 9.

This stateless design means that the authentication works regardless of whether the client requests the data from the end server directly, a proxy, or a load balancer. Therefore, a RESTful API is capable of handling a layered system. Stateless architectures are also ideal in high-volume situations, because they remove the server load caused by the retrieval of session information from a database.

HTTP Methods

In REST, there are four standard ways to interact with a dataset: create, read, update, and delete. These interactions are commonly called *CRUD* operations. REST APIs use the following HTTP methods to perform these operations on the request's resource:

GET Used to retrieve data from a resource. It's the most common request; each time you visit a website in your browser, you make a GET request to the website's address.

POST Used to add a new element to a collection resource. Sending the same POST request multiple times creates a new element for each request, resulting in multiple elements with the same content. When you send an email or submit a web form, your client is usually sending a POST request behind the scenes, because you're creating a new resource in a database.

PUT Used to overwrite or update an existing resource. Sending the same PUT request multiple times creates or overwrites a single element with updated content. For example, when you re-upload a picture on Instagram or Facebook, you might send a PUT request.

PATCH Used to partially update an existing resource. Unlike with PUT, you're sending only the data that differs from the current dataset. Hence, it's a smaller and more performant operation. For example, an update to your profile on a social media page might be done with a PATCH request.

DELETE Used to delete a resource (for example, to remove a picture on Instagram).

REST API requests suffer from the same performance implications as do all HTTP requests. Developers must consider critical factors such as network bandwidth, latency, and server load. While the application usually can't influence the network latency or user bandwidth, it can increase its performance by caching the requests and responding with the previously cached results.

In general, the recommended approach is to cache requests aggressively. By avoiding additional server requests, we can speed up our application significantly. Unfortunately, not all HTTP requests are cacheable. The

responses for GET requests are cacheable by default, but PUT and DELETE answers aren't cacheable at all, because they don't guarantee a predictable response. Between two similar PUT requests, a DELETE request might have deleted the resource, or vice versa. POST and PATCH request responses can, in theory, be cached if the response provides an Expire header or a Cache-Control header and your subsequent calls are GET requests to the same resource. Still, servers frequently won't cache those two types.

Working with REST

Let's practice working with REST by taking a look at our fictional weather service. Say we read the API contract and see that an authorized user can receive and update datasets from the service by using its public REST API. The API returns JSON data, the server's URL is <https://www.usemodernfullstack.dev>, and the endpoint at `/api/v2/weather/{zipcode}` accepts GET and PUT requests. In this section, we walk through the requests and responses for getting the current weather data for a specific ZIP code with a GET request to the API, as well as for updating the stored weather data with a PUT request.

Reading Data

To receive the weather for your location, you might make a GET request containing the ZIP code 96815 and an authorization token. We can make such a GET request with a command line tool like cURL, which should be part of your system. If necessary, you can install it from <https://curl.se>. A typical cURL request looks like this:

```
$ curl -i url
```

The `-i` flag displays the header details we are interested in. We can set the HTTP method with the `-X` flag and send an additional header with the `-H` flag. Use the escape character to send a multiline command (`\` on macOS and `^` on Windows). Avoid adding a space character behind the escape character. If you're curious, try using cURL to query one of the API endpoints in the app you created in Exercise 5 on page 89. A cURL call for a GET request to the weather API `v2/weather/{zipcode}` at <https://www.usemodernfullstack.dev/api> would look like this:

```
$ curl -i \  
-X GET \  
-H "Accept: application/json" \  
-H "Authorization: Bearer 83dedad0728baaef3ad3f50bd05ed030" \  
https://www.usemodernfullstack.dev/api/v2/weather/96815
```

We make this request to the API endpoint `v2/weather/{zipcode}` on the server at <https://www.usemodernfullstack.dev/api>. The ZIP code is included in the URL. We set the return format to JSON in the Accept header and pass an

access token in the Authorization header. Because this is an example API, it accepts any token; if one is not supplied, the API returns a status code of *401*.

Here is what the API's response to our GET request looks like:

```
HTTP/2 200
content-type: application/json ; charset=utf-8
access-control-allow-origin: *

{"weather":"sunny","tempC":"25","tempF":"77","friends":["96814","96826"]}
```

The API responds with an HTTP status code of *200*, indicating that the request was successful. We asked for a JSON response, and the content-type header confirms that the response data is indeed of that type.

The Access-Control-Allow-Origin header, which we discussed in Chapter 5, here allows access to any domain. With this setting, a browser whose client-side JavaScript wants to access the API will allow these requests regardless of the website's domain. Without the CORS header, the browser would block the request and the script's access to the response and instead throw a CORS error.

Finally, we see that the response's body contains a JSON string with the API response.

Updating Data

Now imagine that you want to add display data from your neighborhood (96814) and the adjacent one (96826) to your website. Unfortunately, these ZIP codes aren't yet available on the API. Luckily, because it's open source, we can hook up our own weather station and extend the system. Say we've set up our weather sensors and connected them to the API. As soon as the weather changes, we add the dataset to it.

Here is the PUT request we make to update the weather for the ZIP code 96814. PUT requests store data in the request body; therefore, we use the *-d* flag in the cURL command to send the encoded JSON:

```
$ curl -i \
  -X PUT \
  -H "Accept: application/json" \
  -H "Authorization: Bearer 83dedad0728baef3ad3f50bd05ed030" \
  -H "Content-Type: application/json" \
  -d "{ \"weather\": \"sunny\", \"tempC\": \"20\", \"tempF\": \"68\",
    \"friends\": [ '96815', '96826' ] }" \
  https://www.usemodernfullstack.dev/api/v2/weather/96815
```

We request the same API endpoint, */api/v2/weather/*, but replace the GET method with PUT, because we don't want to get the data from the database; instead, we want to add data. We use the Content-Type header to tell the API provider that the payload in the request body is a JSON string. The API updates the dataset and responds with a status code of *200* and a JSON object with additional status details:

```
HTTP/2 200
content-type: application/json ; charset=utf-8
access-control-allow-origin: *
{"status":"ok"}
```

You can learn more about RESTful APIs at <https://restfulapi.net>, which covers more specific topics, such as compression and security models, and guides you through designing your own RESTful APIs. Now let's turn our attention to GraphQL, a different, more advanced type of API.

GraphQL APIs

Unlike REST, GraphQL isn't merely an architectural pattern. It's a complete, open source data query and manipulation language for APIs. It's also the most popular REST alternative in full-stack web development, used by Airbnb, GitHub, PayPal, and many other companies. In fact, 10 percent of the top 10,000 sites reportedly use GraphQL. This section covers only certain of its features but should give you a solid understanding of GraphQL principles.

NOTE

Despite its name, GraphQL doesn't require the use of a graph database like Neo4j. We can use it to query any data source connected to the GraphQL server, including common databases such as MySQL and MongoDB.

Like REST, GraphQL APIs operate over HTTP. However, a GraphQL implementation exposes only a single API endpoint, typically called `/graphql`, for accessing all resources and performing all CRUD operations. By contrast, REST has one dedicated endpoint per resource.

Another difference is that we connect to the GraphQL server by using POST requests exclusively. Rather than using HTTP methods to define a desired CRUD operation, we use queries and mutations in the POST request body. *Queries* are read operations, while *mutations* are operations for creating, updating, and deleting data.

And unlike REST, which relies on standard HTTP status codes, GraphQL returns `500`, that is, an *Internal Server Error*, when an operation cannot be executed at all. Otherwise, the response uses `200` even if there are problems with the queries or mutations. The reason for this is that the resolver might have partially executed before encountering an issue. Keep this in mind when deploying a GraphQL API in production. Many standard operational practices and tools may need to change to account for this behavior.

The Schema

A GraphQL API defines the available queries and mutations in its schema, which is equivalent to the REST API's specification. Also called a *typedef*, the schema is written in the Schema Definition Language (SDL). SDL's core elements are *types*, which are objects that contain typed *fields* defining their properties, and optional *directives* that add additional information, for example, to specify caching rules for queries or mark fields as deprecated.

Listing 6-2 shows a GraphQL schema for our fictional weather API, which returns the weather data for a location.

```
export const typeDefs = gql`

  type LocationWeatherType {
    zip: String!
    weather: String!
    tempC: String!
    tempF: String!
    friends: [String]!
  }

  input LocationWeatherInput {
    zip: String!
    weather: String
    tempC: String
    tempF: String
    friends: [String]
  }

  type Query {
    weather(zip: String): [LocationWeatherType]!
  }

  type Mutation {
    weather(data: LocationWeatherInput): [LocationWeatherType]!
  }
`;
```

Listing 6-2: The GraphQL schema for the weather API

You should notice that the schema is a tagged template literal, which you learned about in Chapter 2. We begin by describing custom GraphQL object types. These object types represent the data that the API returns. They are similar to the custom types we defined in TypeScript. A type has a name and can implement an interface. Each of these custom object types contains fields, which have a name and a type. GraphQL has the built-in scalar types `Int`, `Float`, `String`, `Boolean`, and `ID`. Exclamation marks (!) denote non-nullable fields, and lists within square brackets ([]) indicate arrays.

The first custom object type is `LocationWeatherType`, which describes the location information for a weather query. Here we use the `String!` expression to mark the `ZIP` field as non-nullable. Hence, the GraphQL service always returns a value for this field. We define a `friends` field as an array of strings to represent related weather stations by ZIP code. It is also non-nullable, so it will always contain an array (with zero or more items) when added to the return values. The `String` inside the `friends` array ensures that every item will be a string.

Then we define the input type object for our first mutation. These types are necessary for mutations, and they represent the input received from the API's consumer. Because consumers should be able to pass in only the fields they'd like to update, we omit the exclamation marks to make the

fields optional. In GraphQL, we need to define input objects and types for the return value separately, with the built-in types. Unlike in TypeScript, we can't use generic custom types.

The schema also defines the query and mutation functions. These are the operations that consumers can send to the API. The weather query takes a ZIP code as a parameter and always returns an array of `WeatherLocationType` objects. The weather mutation takes a `WeatherLocationInput` parameter and always returns the modified `WeatherLocationType` object.

Our schema doesn't contain any directives, and we won't go deep into their syntax in this chapter. However, one reason to use directives is for caching. Because GraphQL queries use POST, which isn't cacheable using the default HTTP cache, we must implement caching manually, on the server side. We can configure caching statically on the type definitions with the directive `@cacheControl` or dynamically, in the resolver functions, with `cacheControl.setCacheHint`.

The Resolvers

In GraphQL, the *resolvers* are the functions that implement the schema. Each resolver function maps to a field. Query resolvers implement the reading of data, whereas mutation resolvers implement the creation, updating, and deletion of data. Together they provide complete CRUD functionality.

To understand how resolvers work, you can think of each GraphQL operation as a tree of nested function calls. In such an *abstract syntax tree* (AST), each part of the operation represents a node. For example, consider a complex, nested GraphQL query, which asks for the location's current weather, as well as the weather of all its neighbors. Our GraphQL schema for this example looks like Listing 6-3.

```
export const typeDefs = gql`

  type FriendsType {
    zip: String!
    weather: String!
  }

  type LocationWeatherType {
    zip: String!
    weather: String!
    tempC: String!
    tempF: String!
    friends: [FriendsType]!
  }

  type Query {
    weather(zip: String!): [LocationWeatherType]!
  }
`;
```

Listing 6-3: The GraphQL schema for the nested GraphQL query example

In the schema for the example, we replace the content of the friends array. Instead of a simple string, we want it to contain an object with a ZIP code and weather information. Therefore, we define a new FriendsType and use this type for the array items.

Listing 6-4 defines the complex example query.

```
query GetWeatherWithFriends {  
  weather(zip: "96815") {  
    weather  
    friends {  
      weather  
    }  
  }  
}
```

Listing 6-4: The nested GraphQL query

This query takes the zip parameter 96815 and then returns its weather property, as well as all of its friends' weather properties, as strings. But how does the query work under the hood?

Figure 6-1 shows the resolver chain and corresponding AST. The GraphQL server would first parse the query into this structure and then validate the AST against the type-definition schema to ensure that the query can be executed without running into logical problems. Finally, the server would execute the query.

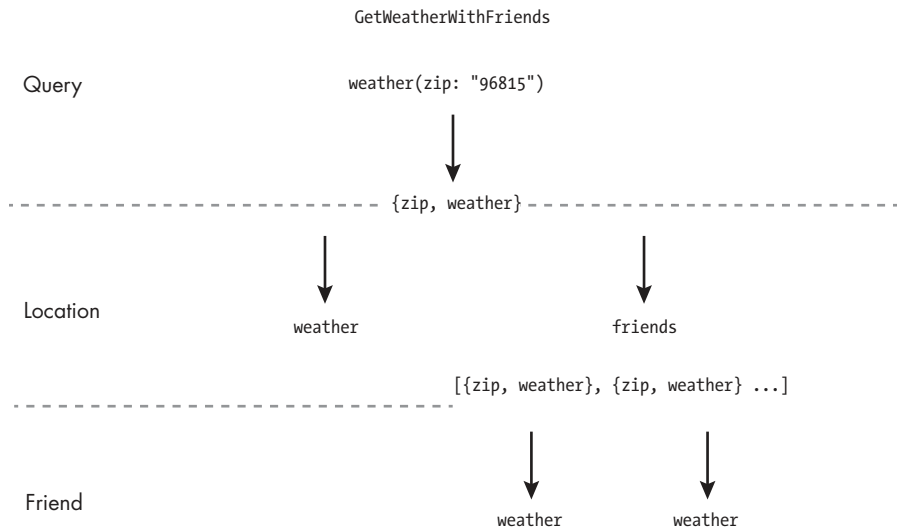


Figure 6-1: Querying the GraphQL AST

Let's examine the resolver chain for the query. The `Query.weather` function takes one argument, the ZIP code, and returns the location object for this ZIP code. Then the server continues along each branch separately. For the weather in the query, it returns the weather property of the location

object, `Location.weather`, at which point the branch ends. The second part of the query, which asks for all friends from the location object and their weather properties, runs the `Location.Friends` query and then returns the `Friends.weather` property for each result. The resolver object of each step contains the result returned by the parent field's resolver.

Let's return to our weather schema and define the resolvers. We'll keep these simple. In Listing 6-5, you can see that their names match those defined in the schema.

```
export const resolvers = {
  Query: {
    weather: async (_, param: WeatherInterface) => {
      return [
        {
          zip: param.zip,
          weather: "sunny",
          tempC: "25C",
          tempF: "70F",
          friends: []
        }
      ];
    },
  },
  Mutation: {
    weather: async (_, param: { data: WeatherInterface }) => {
      return [
        {
          zip: param.data.zip,
          weather: "sunny",
          tempC: "25C",
          tempF: "70F",
          friends: []
        }
      ];
    },
  },
};
```

Listing 6-5: The GraphQL resolvers for the weather API

We first define async functions for the query and mutation properties and assign the object to the `const resolvers`. Each takes two parameters. The first one represents the previous resolver object in the resolver chain. We aren't using a nested or complex query; hence, here it is always undefined. For this reason, we use the `any` type to avoid a TypeScript error and use the underscore (`_`) convention you learned in Chapter 3 to mark it as unused. The second parameter is an object containing the data passed to the function on invocation. For the `weather` query and the `weather` mutation, it is an object that implements the `WeatherInterface`.

For now, both functions ignore this parameter for the most part, using only the `zip` property to reflect the input. Also, they return a static JSON

object similar to the REST API we created in the previous listings. The static data is just a placeholder, which we'll replace with the result from our database queries later. The response honors the API contract we defined in the GraphQL schema, as this data consists of arrays with weather location datasets.

Comparing GraphQL to REST

We've already implemented RESTful APIs in our Next.js application, and as you saw in this chapter, REST is fairly simple to work with. You might be wondering why you'd even consider using GraphQL. Well, GraphQL solves two problems common in REST APIs: over-fetching and under-fetching.

Over-Fetching

When a client queries a REST endpoint, the API always returns the complete dataset for that endpoint. This means that, often, the API delivers more data than necessary, a common performance problem called *over-fetching*. For example, our example RESTful weather API at `/api/v2/weather/zip/96815` delivers all weather data for a ZIP code even if all you need is the temperature in Celsius. You'd then need to manually filter the results. In GraphQL, the API requests explicitly define the data they want returned.

Let's look at an example to see how GraphQL lets us keep the API response data to a minimum. The following GraphQL query returns only the temperature in Celsius for the location with the ZIP code 96815:

```
query Weather {  
  weather(zip: "96815") {  
    tempC  
  }  
}
```

In GraphQL, we send the query as a JSON string with the POST request's data:

```
$ curl -i \  
  -X POST \  
  -H "Accept: application/json" \  
  -H "Authorization: Bearer 83dedad0728baaef3ad3f50bd05ed030" \  
  -H "Content-Type: application/json" \  
  -d '{"query": "\nquery Weather {\n  weather( zip: \"96815\") {\n    tempC\n  }\n}" }' \  
  https://www.usemodernfullstack.dev/api/graphql
```

We consume the API with a POST request to the `/api/graphql` endpoint, then set the Content-Type header and Accept header to JSON to explicitly tell the API that we're sending a JSON object in the request body and expect a JSON response. We set the access token in the Authorization header, as in a

RESTful request. The POST body contains the query for the weather data, and the `\n` control characters indicate the newlines in the GraphQL query. As defined in the contract, the query expects a parameter, `zip`, for which we pass in the ZIP code 96815. In addition, we request that the API return only the `tempC` field of the weather node.

Here is the response from the GraphQL API:

```
HTTP/2 200
content-type: application/json ; charset=utf-8
access-control-allow-origin: *

{"data":{"weather":[{"tempC":"25C"}]}}
```

The API responds with a status code of `200`. We specified in the request's query that we are interested in only the requested field `tempC` of the weather object, so this is what we received. The API doesn't return the ZIP code, temperature in Fahrenheit, weather string, or friends array.

Under-Fetching

On the other hand, a REST dataset might not contain all the data you need, requiring you to send follow-up requests. This problem is called *under-fetching*. Imagine that your friends also have weather stations and that you want to get the current weather at their ZIP codes. The RESTful weather API returns an array with related ZIP codes (`friends`). However, you'd need to make additional requests for each ZIP code to receive their weather information, potentially causing performance issues.

GraphQL treats datasets as nodes in a graph, with relationships between them. Therefore, extending a single query to receive related data is pretty simple. Our example GraphQL server's resolvers are set up to fetch additional data about friends if the request's query contains the `friends` field. We define the GraphQL query as follows:

```
query Weather {
  weather( zip: "96815") {
    tempC
    friends {
      tempC
    }
  }
}
```

The following shows an example request that fetches all related nodes through the `friends` array. Again, we define the return data and query the friends only for the field `tempC`:

```
$ curl -i \
-X POST \
-H "Accept: application/json" \
-H "Authorization: Bearer 83dedad0728baaef3ad3f50bd05ed030" \
```

```
-H "Content-Type: application/json" \  
-d '{"query": "query Weather {\n  weather( zip: \"96815\")  
    {\n      tempC\n      friends {\n        tempC\n      }\n    }\"}' \  
https://www.usemodernfullstack.dev/api/graphql
```

The POST body contains the query for weather data pertaining to the 96815 ZIP code in one line and asks for the tempC field, as in the previous request. To extend the query, we add a sub-selection on the friends field. Now GraphQL traverses the related nodes and their fields and returns the tempC field of the nodes whose ZIP codes match the ones in the 96815 node's friends array.

Here is the response from the GraphQL server. We see that it contains data from the related nodes:

```
HTTP/2 200  
content-type: application/json ; charset=utf-8  
access-control-allow-origin: *  
  
{ "data": { "weather": [ { "tempC": "25C", "friends":  
  [ { "tempC": "20C" }, { "tempC": "30C" } ] } ] } }
```

As you've discovered, GraphQL lets us easily extend queries by adjusting the data in the request.

Exercise 6: Add a GraphQL API to Next.js

Let's rework our weather application's API to use GraphQL. To do so, we must first add GraphQL to the project. GraphQL isn't a pattern but an environment that consists of a server and a query language, both of which we must add to Next.js.

We'll install the stand-alone Apollo server, one of the most popular GraphQL servers, which also provides a Next.js integration. Open your terminal and navigate to the refactored application you built in Chapter 5. In the directory's top level, next to the *package.json* file, execute this command:

```
$ npm install @apollo/server @as-integrations/next graphql graphql-tag
```

This command also installs the GraphQL language and the GraphQL tag modules we'll need.

Creating the Schema

As we discussed, every GraphQL API starts with a schema definition. Create a folder called *graphql* next to the *pages* folder in the Next.js directory. This is where we'll add all GraphQL-related files.

Now create a file called *schema.ts* and paste the code you wrote back in Listing 6-2. We've already defined and discussed the type definition used here. Simply add one line to the top of the file:

```
import gql from "graphql-tag";
```

This line imports the `qql` tagged template literal we use to define the schema.

Adding Data

We want our API to return different data depending on the parameters and properties of the queries sent to it. Therefore, we need to add datasets to our project. GraphQL can query any database, even static JSON data. So let's implement a JSON dataset. Create the file *data.ts* inside the *graphql* directory and add the code from Listing 6-6.

```
export const db = [
  {
    zip: "96815",
    weather: "sunny",
    tempC: "25C",
    tempF: "70F",
    friends: ["96814", "96826"]
  },
  {
    zip: "96826",
    weather: "sunny",
    tempC: "30C",
    tempF: "86F",
    friends: ["96814", "96814"]
  },
  {
    zip: "96814",
    weather: "sunny",
    tempC: "20C",
    tempF: "68F",
    friends: ["96815", "96826"]
  }
];
```

Listing 6-6: The *graphql/data.ts* file for the GraphQL API

This JSON defines three weather locations and their properties. A consumer will be able to query our API for these datasets.

Implementing Resolvers

Now we can define our resolvers. Add the file *resolvers.ts* to the *graphql* directory and paste in the code from Listing 6-7. This is similar to the code we previously discussed when we introduced resolvers, but instead of returning the same static JSON object to the consumer, we query our new dataset.

```
import { db } from "../data";

declare interface WeatherInterface {
  zip: string;
  weather: string;
  tempC: string;
```

```

    tempF: string;
    friends: string[];
  }

export const resolvers = {
  Query: {
    weather: async (_, param: WeatherInterface) => {
      return [db.find((item) => item.zip === param.zip)];
    },
  },
  Mutation: {
    weather: async (_, param: { data: WeatherInterface }) => {
      return [db.find((item) => item.zip === param.data.zip)];
    },
  },
};

```

Listing 6-7: The graphql/resolvers.ts file for the GraphQL API

We import the array of JSON objects we created earlier and define an interface for the resolvers. The query resolver finds an object by using the ZIP code passed to it and returns it to the Apollo server. The mutation does the same, except that the parameter structure is slightly different: it is accessible through the data property. Alas, we can't actually change the data by using the mutation, as the data is a static JSON file. We've implemented the mutation here for illustration purposes only.

Creating the API Route

The Apollo GraphQL server exposes one endpoint, *graphql/*, which we'll implement now. Create a new file, *graphql.ts*, in the *api* folder and add the code from Listing 6-8. This code initializes the GraphQL server and adds a CORS header so that we can access the API from different domains and use the built-in GraphQL sandbox explorer to play with GraphQL later. You saw this header in the previous cURL responses.

```

import { ApolloServer } from "@apollo/server";
import { startServerAndCreateNextHandler } from "@as-integrations/next";
import { resolvers } from "../graphql/resolvers";
import { typeDefs } from "../graphql/schema";
import { NextApiHandler, NextApiRequest, NextApiResponse } from "next";

//@ts-ignore
const server = new ApolloServer({
  resolvers,
  typeDefs
});

const handler = startServerAndCreateNextHandler(server);

const allowCors =
  (fn: NextApiHandler) => async (req: NextApiRequest, res: NextApiResponse) => {
    res.setHeader("Allow", "POST");

```



```

res.setHeader("Access-Control-Allow-Origin", "*");
res.setHeader("Access-Control-Allow-Methods", "POST");
res.setHeader("Access-Control-Allow-Headers", "*");
res.setHeader("Access-Control-Allow-Credentials", "true");

if (req.method === "OPTIONS") {
  res.status(200).end();
}
return await fn(req, res);
};

export default allowCors(handler);

```

Listing 6-8: The `api/graphql.ts` file, which creates the API entry point for GraphQL

This code is all we need to create the GraphQL entry point. First we import the necessary modules, including our GraphQL schema and the resolvers, both of which we created previously. Then we initialize a new GraphQL server with typedefs and resolvers.

We start the server and continue by creating the API handler. To do this, we use the Next.js integration helper to start the server and return the Next.js handler. The integration helper connects the serverless Apollo instance to the Next.js custom server. Before we define the default export as an async function that takes the API's request and response objects as parameters, we create a wrapper to add the CORS headers to the request. The first block inside the function sets up the CORS headers, and we limit the allowed request to POST requests. We need the CORS headers here to make our GraphQL API publicly available. Otherwise, we wouldn't be able to connect to the API from a website running on a different domain or even use the server's built-in GraphQL sandbox.

Part of the CORS setup here is that we immediately return `200` for any OPTIONS requests. The CORS patterns use OPTIONS requests as preflight checks. Here the browser requests only headers, and then checks the response's CORS headers to verify that the domain from which it calls the API is allowed to access the resource before making the actual request.

However, our Apollo server allows only POST and GET requests and would return `405: Method Not Allowed` for the preflight OPTIONS request. So, instead of passing this request to the Apollo server, we end the request and return `200` with the previous CORS headers. The browser should then proceed with the CORS pattern. Finally, we start the server and create the API handler on the desired path, `api/graphql`.

Using the Apollo Sandbox

Start your Next.js server with `npm run dev`. You should see the Next.js application running on `http://localhost:3000`. If you navigate to the GraphQL API at `http://localhost:3000/api/graphql`, you'll find the Apollo sandbox interface for querying the API, as in Figure 6-2.

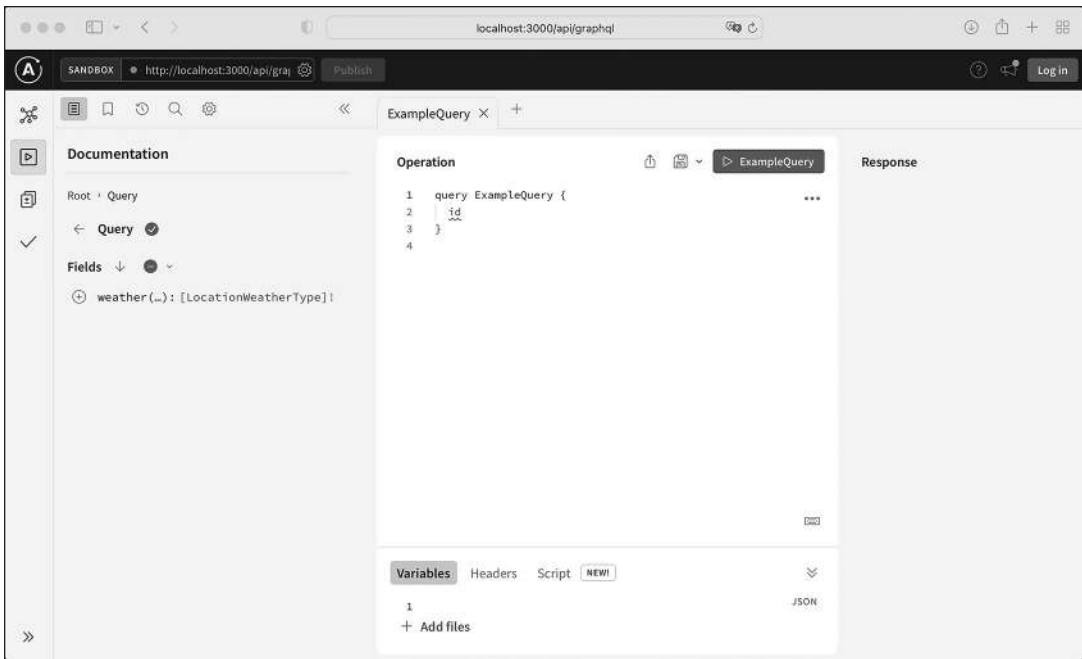


Figure 6-2: The Apollo sandbox's API querying interface

In the Documentation pane on the left side, we see the available queries as fields of the query object we defined earlier. As expected, we see the `Weather` query here, and when we click it, a new query appears in the Operation pane in the middle. At the same time, the interface changes, and we see the available arguments and fields. Clicking each provides more information. Using the plus (+) button, we can add fields to the Query pane and run them against the data.

Try creating a `Weather` query that returns the `zip` and `weather` properties. This query requires a `ZIP` code as an argument; add it through the user interface on the left-hand side, and then add the `ZIP` code `96826` as a string to the JSON object in the variables section of the lower pane. Now run the query by clicking the **Weather** button at the top of the Operation pane. You should receive the result for this `ZIP` code in the Response pane on the right as JSON. Compare your screen with Figure 6-3.

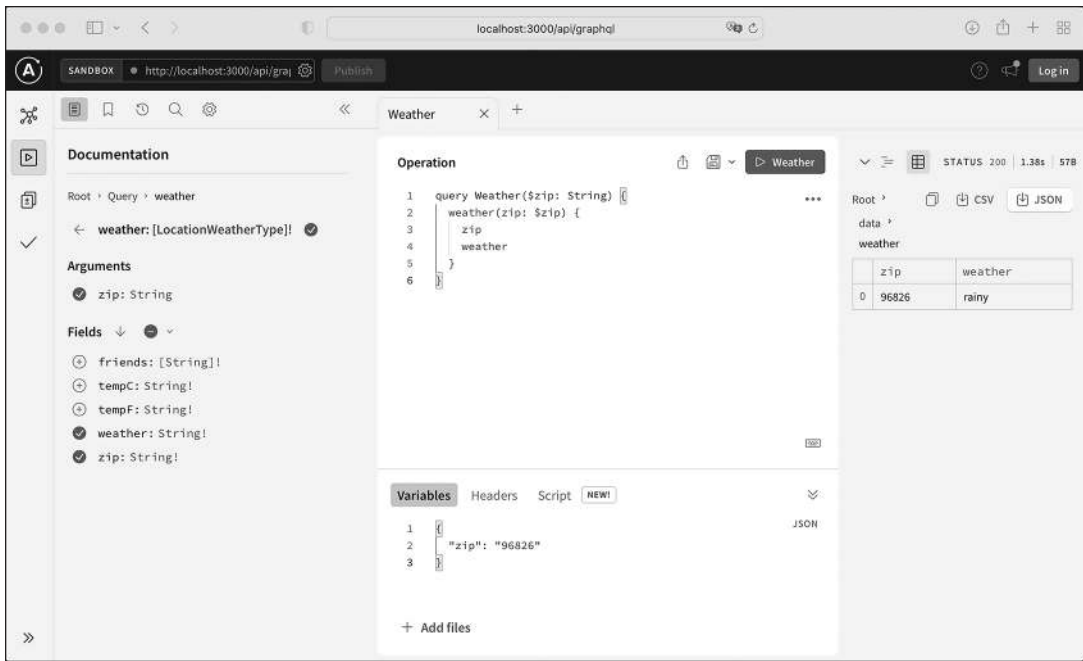


Figure 6-3: The GraphQL query and response from the server

Play around with crafting queries, accessing properties, and creating errors with invalid arguments to get a feel for GraphQL before moving on to the next chapter.

Summary

This chapter explored RESTful and GraphQL web APIs and their role in full-stack development. Although we used a REST design in previous chapters, you should now be familiar with the concept of stateless servers, as well as the five HTTP methods for performing CRUD operations in REST. You also practiced working with a public REST API to read and update data, then evaluated its requests and responses.

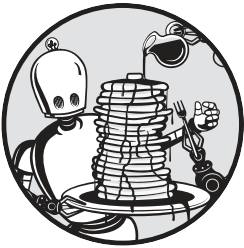
GraphQL APIs require a bit more work to implement, but they reduce the over-fetching and under-fetching issues often experienced in REST. You learned to define the API contract with a schema and implement its functionality with resolvers. Then you queried an API and defined the dataset to return in the request.

Finally, you added a GraphQL API to your existing Next.js application by adding the Apollo server to it. You should now be able to create your own GraphQL API and consume third-party resources. To learn more about GraphQL, I recommend the tutorials at <https://www.howtographql.com> and the official GraphQL introduction at <https://graphql.org/learn/>.

In the next chapter, you'll explore the MongoDB database and Mongoose, an object data modeling library, for storing data.

7

MONGODB AND MONGOOSE



Most applications rely on a database management system, or *database* for short, to organize and grant access to a collection of datasets. In this chapter, you'll work with the MongoDB non-relational database and Mongoose, its accompanying object mapper.

Because MongoDB returns data as JSON and uses JavaScript for database queries, it provides a natural choice for full-stack JavaScript developers. In the following sections, you'll learn how to create a Mongoose model through which you can query your database, simplify your interactions with MongoDB, and craft middleware that connects your frontend to your backend database. You'll also write service functions to implement the four CRUD operations on the database.

In Exercise 7 on page 125, you'll add a database to the GraphQL API you created in Chapter 6, replacing its current static datastore.

How Apps Use Databases and Object-Relational Mappers

An app needs a database to store and manipulate data. So far in this book, our app’s APIs returned only predefined datasets, saved in files, that couldn’t change. We used parameters in our requests to add to the dataset but couldn’t store the data between different API calls (called *persisting* the data). If we wanted to update the app’s weather information, for example, we’d need a database to persist the data so that the next API call could read it. In full-stack development, we commonly use databases to store user-related data. Another example of a database is the one that your email client uses to store your messages.

To work with a database, we first need to connect to it and authenticate with it. Once we have access to the data, we can execute queries to ask for certain datasets. The query returns the results containing data that our app can display or use in some other way. How each of these steps works in practice depends on the specific database in use.

Querying the data by using the database’s API tends to be clumsy because it usually requires a good amount of boilerplate code, even to simply establish and maintain the connection. Hence, we often use an *object-relational mapper* or *object data modeling tool*, which simplifies working with the databases by abstracting some of the details. For example, the Mongoose object data modeling tool for MongoDB handles database connections for us, saving us from having to check for an open database connection during each interaction.

Mongoose also makes it easier to handle the fact that MongoDB runs on a separate database server. Working with distributed systems requires making asynchronous calls, which you learned about in Chapter 2. With Mongoose, we can access the data with an object-oriented `async/await` interface instead of using clumsy callback functions.

In addition, MongoDB is schema-less; it doesn’t require us to predefine and strictly adhere to a schema. While convenient, this flexibility is also a common source of errors, especially in large-scale applications or projects with a rotating cast of developers. In Chapter 3, we discussed the benefits of adding types to JavaScript by using TypeScript. Mongoose types and verifies the integrity of MongoDB’s data models similarly, as you’ll discover in “Defining a Mongoose Model” on page 118.

Relational and Non-Relational Databases

Databases can organize data in several ways, which fall into two main categories: relational and non-relational. *Relational databases*, such as MySQL and PostgreSQL, store data in one or more tables. You can think of these databases as resembling Excel spreadsheets. As in Excel, each table has a unique name and contains columns and rows. The columns define properties, such as the data type, for all data stored in the column, and the rows contain the actual datasets, each of which is identified by a unique ID. Relational databases use some variation of Structured Query Language (SQL) for their database operations.

MongoDB is a *non-relational database*. Unlike traditional relational databases, it stores data as JSON documents instead of tables and doesn't use SQL. Sometimes termed *NoSQL*, non-relational databases can store data in many different formats. For example, the popular NoSQL databases Redis and Memcached use key-value storage, which makes them highly performant and easily scalable. Thus, they're often used as in-memory caches. Another NoSQL database, Neo4j, is a *graph database* that uses graph theory to store data as nodes, a concept we mentioned in Chapter 6. These are just a few examples of non-relational databases.

MongoDB is the most widely used *document database*; instead of tables, rows, and columns, it organizes data in collections, documents, and fields. The *field* is the smallest unit in the database. It defines the data type and additional properties and contains the actual data. You can consider it the rough equivalent of a column in a SQL table. *Documents*, which are made of fields, are like rows in a SQL table. We sometimes call them records, and MongoDB uses *BSON*, a binary representation of a JSON object, to store them. A *collection* is roughly equivalent to a SQL table, but instead of rows and columns, it aggregates documents.

Because non-relational databases can store data in different formats, each database uses a specific, optimized query language for CRUD operations. These low-level APIs focus on accessing and manipulating the data, and not necessarily on the developer experience. By contrast, object-relational mappers provide a high-level abstraction with a clean and simplified interface to the query language. So, while MongoDB has the MongoDB Query Language (MQL), we'll use Mongoose to access it.

Setting Up MongoDB and Mongoose

Before you start using MongoDB and Mongoose, you must add them to your sample project. For the sake of simplicity, we'll use an in-memory implementation of MongoDB rather than install and maintain a real database server on our machines. This is appropriate for testing the chapter's examples, but not for deploying an actual application, as it does not persist the data between restarts. You'll gain experience setting up a real MongoDB server when you build the Food Finder application in Part II. Chapter 11 will show you how to use a pre-built Docker container that contains the MongoDB server.

Run this command in the root directory of the refactored Next.js app from Chapter 6:

```
$ npm install mongodb-memory-server mongoose
```

Then create two new folders in the root directory, next to the *package.json* file: one for the Mongoose code, called *mongoose*, with subfolder *weather*, and one called *middleware*, which will hold the necessary middleware.

Defining a Mongoose Model

In order to verify the integrity of our data, we must create a schema-based Mongoose *model*, which acts as a direct interface to a MongoDB collection in a database. All interactions with the database will happen through the model. Before we create the model, though, we need to create the schema itself, which defines the structure of the database's data and maps the Mongoose instance to the documents in the collection.

Our Mongoose schema will match the schema created for the GraphQL API in Chapter 6. That's because we'll connect the GraphQL API to the database in Exercise 7 on page 125, allowing us to replace the static JSON object with datasets we queried from the database.

The Interface

Before writing the Mongoose model and schema in TypeScript, let's declare a TypeScript interface. Without a matching interface, we won't be able to type the model or schema for TSC, and the code won't compile. Paste the code shown in Listing 7-1 into the *mongoose/weather/interface.ts* file.

```
export declare interface WeatherInterface {  
  zip: string;  
  weather: string;  
  tempC: string;  
  tempF: string;  
  friends: string[];  
};
```

Listing 7-1: The interface for the Mongoose weather model

The code is a regular TypeScript interface with properties matching the GraphQL and Mongoose schemas.

The Schema

Listing 7-2 shows the Mongoose schema. Its top-level properties represent the fields in the document. Each field has a type and a flag indicating whether it is required. Fields can also have additional optional properties, such as custom or built-in validators. Here we use the built-in required validator; other common built-in validators are *minlength* and *maxlength* for strings, and *min* and *max* for numbers. Add the code to the *mongoose/weather/schema.ts* file.

```
import { Schema } from "mongoose";  
import { WeatherInterface } from "../interface";  
  
export const WeatherSchema = new Schema<WeatherInterface>({  
  zip: {  
    type: "String",  
    required: true,  
  },  
});
```



```

    weather: {
      type: "String",
      required: true,
    },
    tempC: {
      type: "String",
      required: true,
    },
    tempF: {
      type: "String",
      required: true,
    },
    friends: {
      type: ["String"],
      required: true,
    },
  },
});

```

Listing 7-2: The schema for the Mongoose weather model

We use an object passed to the schema constructor to create the schema and set `WeatherInterface` as its `SchemaType`. Therefore, we import the `Schema` function from the *mongoose* package and the interface we created previously.

Like TypeScript, which adds custom types to JavaScript, Mongoose casts each property to its associated *SchemaType*, which provides the configuration of the model. The available types are a mixture of built-in JavaScript types, like `Array`, `Boolean`, `Date`, `Number`, and `String`, and custom types, like `Buffer` and `ObjectId`, the latter of which refers to the default unique `_id` property that Mongoose adds to each document upon creation. This is similar to the primary key you might know from relational databases.

The weather API we created in Chapter 6 returned an object with four properties: `zip`, `weather`, `tempC`, and `tempF`, each of which is a string. In addition, we have one array of strings in the `friends` property. In this schema, we define the same properties, then export the schema.

The Model

Now that we have a schema, we can create the Mongoose model. This wrapper on the schema will provide access to the MongoDB documents in the collection for all CRUD operations. We write the model in the *mongoose/weather/model.ts* file, whose code is in Listing 7-3. Keep in mind that we haven't yet connected it to the MongoDB database on the server.

```

import mongoose, { model } from "mongoose";
import { WeatherInterface } from "../interface";
import { WeatherSchema } from "../schema";

export default mongoose.models.Weather ||
  model<WeatherInterface>("Weather", WeatherSchema);

```

Listing 7-3: The Mongoose weather model

First we import the Mongoose module and the model constructor from the *mongoose* package, as well as the interface and the schema we created earlier. Then we set up the Weather model, using *WeatherInterface* to type it. We pass it two parameters: the model's name, *Weather*, and the schema, which defines the model's internal data structure. Mongoose binds the newly created model to our MongoDB instance's collection. The *Weathers* collection resides in the *Weather* database, both of which Mongoose creates. Note that we need to check for an existing *Weather* model on *mongoose.models* before creating a new one; otherwise, Mongoose will throw an error. We export the model so that we can use it in our following modules.

The Database-Connection Middleware

Several times in this book so far, we've mentioned that full-stack development covers an application's frontend, backend, and middleware, which is often also referred to as "application glue." Now it's time to create our first dedicated middleware.

This middleware will open a connection to the database, then use Mongoose's asynchronous helper function to maintain that connection. Next, it will map Mongoose's models to the MongoDB collections so that we can access them through Mongoose. Conveniently, the connection helper will buffer the operations and reconnect to the database if necessary, so we don't need to handle connectivity issues by ourselves. Paste the code from Listing 7-4 into the *middleware/db-connect.ts* file.

```
import mongoose from "mongoose";
import { MongoMemoryServer } from "mongodb-memory-server";

async function dbConnect(): Promise<any | String> {
  const mongoServer = await MongoMemoryServer.create();
  const MONGOIO_URI = mongoServer.getUri();
  await mongoose.disconnect();
  await mongoose.connect(MONGOIO_URI, {
    dbName: "Weather"
  });
}

export default dbConnect;
```

Listing 7-4: The Mongoose middleware

We import the *mongoose* package and the *mongodb-memory-server* database. The async function *dbConnect*, which we define and then export, manages the connection to the database server through the *mongoose.connect* function. We create an instance of the *MongoMemoryServer* to persist our data in memory rather than use a real database server, as discussed. Then we store the connection string in the constant *MONGOIO_URI*. Because we are using the in-memory server, this string is dynamic, but for a remote database, it would be a static string representing the database's server address.

Then we close all existing connections and use Mongoose to open a new connection. The Mongoose models are already mapped and available, so we're ready to perform our first queries.

Querying the Database

Now it's time to write database queries. Instead of sprinkling these queries around your application code or writing them directly in the GraphQL resolvers, you should extract them as services.

A *service* is a function that performs the actual CRUD operations on the Mongoose model and returns the result. Each GraphQL resolver can then call a service function, and all internal database access should happen through these functions. Moreover, each service should be responsible for only one specific CRUD operation. Mongoose automatically queues the commands and executes them, maintains the connection, and then processes the queue as soon as there is a connection to the database.

This section introduces service functions and basic Mongoose commands. However, it isn't a complete reference. When you start working with Mongoose on your own projects, look up all the functions you'll need in the Mongoose documentation.

Creating a Document

The first and most basic operation is the “create” operation. It is conveniently called `mongoose.create` and, fortunately, we can use it to both create and update a dataset. That's because Mongoose automatically creates a new database entry, or document, if the entry doesn't already exist. Hence, we don't need to check whether a dataset exists and then conditionally create it before updating it.

Listing 7-5 shows a basic implementation of a service function that stores a dataset in the database. Place the code in the *mongoose/weather/services.ts* file.

```
import WeatherModel from "../model";
import { WeatherInterface } from "../interface";

export async function storeDocument(doc: WeatherInterface): Promise<boolean> {
  try {
    await WeatherModel.create(doc);
  } catch (error) {
    return false;
  }
  return true;
}
```

Listing 7-5: Creating a document through Mongoose

To store a document, we create and export the async function `storeDocument`, which takes the dataset as the argument. Here we type it

as `WeatherInterface`. Then we call the `create` function on the model and pass the dataset to it. The function will create and insert the document in `WeatherModel`, which is the weather collection in the MongoDB instance. Finally, it returns a Boolean to indicate the status of the operation.

Reading a Document

To implement the “read” operation, we query MongoDB through Mongoose’s `findOne` function. It takes one argument, an object with the properties to look for, and returns the first match. Extend the *mongoose/weather/services.ts* file with the code in Listing 7-6. It defines a `findByZip` function to find and return the first document from the `Weathers` collection whose `zip` property matches the ZIP code passed to the function as a parameter.

```
export async function findByZip(
  paramZip: string
): Promise<Array<WeatherInterface> | null> {
  try {
    return await WeatherModel.findOne({ zip: paramZip });
  } catch (err) {
    console.log(err);
  }
  return [];
}
```

Listing 7-6: Reading data through Mongoose

We add and export the async function `readByZip` to the services in the *services.ts* file. The function takes one string parameter, the ZIP code, and returns either an array with documents or an empty array. Inside the new service function, we call Mongoose’s `findOne` function on the model and pass a filter object, looking for the document whose `zip` field matches the parameter’s value. Finally, the function returns the result or `null`.

Updating a Document

We mentioned that we can use the `create` function to update documents. However, there is also a specific API for this task: `updateOne`. It takes two arguments. The first is the filter object, similar to the filter we used with `findOne`, and the second is an object with the new values. You can think of `updateOne` as a combination of the “find” and “create” functions. Extend the *mongoose/weather/services.ts* file with the code from Listing 7-7.

```
export async function updateByZip(
  paramZip: string,
  newData: WeatherInterface
): Promise<boolean> {
  try {
    await WeatherModel.updateOne({ zip: paramZip }, newData);
    return true;
  }
}
```

```
    } catch (err) {
      console.log(err);
    }
    return false;
  }
}
```

Listing 7-7: Updating data through Mongoose

The `updateByZip` function that we add to the services takes two parameters. The first one is a string, `paramZip`, which is the ZIP code we use to query for the document we want to update. The second parameter is the new dataset, which we type as `WeatherInterface`. We call Mongoose's `updateOne` function on the model, passing it a filter object and the latest data. The function should return a Boolean to indicate the status.

Deleting a Document

The last CRUD operation we need to implement is a service to delete a document. For this, we use Mongoose's `deleteOne` function and add the code from Listing 7-8 to the `mongoose/weather/services.ts` file. It is similar to the `findOne` function, except that it directly deletes the query's result. Mongoose queues the operations and deletes the document from the database automatically once there is a connection.

```
export async function deleteByZip(
  paramZip: string
): Promise<boolean> {
  try {
    await WeatherModel.deleteOne({ zip: paramZip });
    return true;
  } catch (err) {
    console.log(err);
  }
  return false;
}
```

Listing 7-8: Deleting data through Mongoose

The async function `deleteByZip` takes one string parameter, `zip`. We use it to query the model and find the document to delete, passing the filter to Mongoose's `deleteOne` function. The function should return a Boolean.

Creating an End-to-End Query

In full-stack development, *end-to-end* typically refers to the ability of data to travel all the way from the app's frontend (or from one of its APIs) through the middleware to the backend, and then all the way back to its original source. For practice, let's create a simple end-to-end example using the `/zipcode` endpoint of our REST API.

We'll modify the API to take the query parameter from the URL, find the weather object for the requested ZIP code in the database, and then

return it, effectively replacing the static JSON response with a dynamic query result. Modify the file *pages/api/v1/weather/[zipcode].ts* to match Listing 7-9.

```
import type { NextApiRequest, NextApiResponse } from "next";
import { findByZip } from "../../../../../mongoose/weather/services";
import dbConnect from "../../../../../middleware/db-connect";
dbConnect();

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse
): Promise<NextApiResponse<WeatherDetailType> | void> {
  let data = await findByZip(req.query.zipcode as string);
  return res.status(200).json(data);
}
```

Listing 7-9: The end-to-end REST API

Notice the modified API handler. We made two major changes to it. First we called `dbConnect` to connect to the database. Then we used the imported `findByZip` service and passed it the query parameter cast to a string type. Instead of the static JSON object as before, we now return the dynamic data that we receive from the service function.

We need to perform one more step before we can receive data in response to the API call: *seeding* the database, or adding initial datasets to it. For simplicity, we use the `storeDocuments` service and seed directly in the `dbConnect` function. Modify the *middleware/db-connect.ts* file to match the code in Listing 7-10, which imports the `storeDocument` service and adds the datasets after establishing the database connection.

```
import mongoose from "mongoose";
import { MongoMemoryServer } from "mongodb-memory-server";
import { storeDocument } from "../../mongoose/weather/services";

async function dbConnect(): Promise<any | String> {
  const mongoServer = await MongoMemoryServer.create();
  const MONGOIO_URI = mongoServer.getUri();
  await mongoose.disconnect();

  let db = await mongoose.connect(MONGOIO_URI, {
    dbName: "Weather"
  });

  await storeDocument({
    zip: "96815",
    weather: "sunny",
    tempC: "25C",
    tempF: "70F",
    friends: ["96814", "96826"]
  });
}
```

```

    await storeDocument({
      zip: "96814",
      weather: "rainy",
      tempC: "20C",
      tempF: "68F",
      friends: ["96815", "96826"]
    });
    await storeDocument({
      zip: "96826",
      weather: "rainy",
      tempC: "30C",
      tempF: "86F",
      friends: ["96815", "96814"]
    });
  }
  export default dbConnect;

```

Listing 7-10: The naive data seeding in the dbConnect function

Now we can perform the end-to-end request. Visit the REST API endpoint in the browser at <http://localhost:3000/api/v1/weather/96815>. You should see the dataset from the MongoDB database as the API response. Try adjusting the query parameter in the URL to another valid ZIP code. You should get another dataset in the response.

Exercise 7: Connect the GraphQL API to the Database

Let's rework our weather application's GraphQL API so that it reads the response data from the database instead of from a static JSON file. The code will look familiar, as we'll use the same patterns as for the REST API example in the preceding section.

First, verify that you've added the MongoDB memory implementation and Mongoose to your project. If not, add them now by following the instructions in "Setting Up MongoDB and Mongoose" on page 117. Next, check that you've created the files in the *middleware* and *mongoose* folders described throughout this chapter and that they contain the code from Listings 7-1 through 7-10.

Now, to connect the GraphQL API to the database, we need to do two things: implement the database connection and refactor the GraphQL resolvers to use its datasets.

Connecting to the Database

To query the database through the GraphQL API, we need to have a connection to the database. As you learned in Chapter 6, all API calls have the same endpoint, */graphql*. This fact will now prove incredibly convenient for us; because all requests have the same entry point, we need to handle the database connection only once. Hence, we open the file *api/graphql.ts* and modify it to match the code in Listing 7-11.

```

import { ApolloServer } from "@apollo/server";
import { startServerAndCreateNextHandler } from "@as-integrations/next";
import { resolvers } from "../../graphql/resolvers";
import { typeDefs } from "../../graphql/schema";
import { NextApiHandler, NextApiRequest, NextApiResponse } from "next";
import dbConnect from "../../middleware/db-connect";
//@ts-ignore
const server = new ApolloServer({
  resolvers,
  typeDefs
});

const handler = startServerAndCreateNextHandler(server);

const allowCors = (fn: NextApiHandler) =>
  async (req: NextApiRequest, res: NextApiResponse) => {
    res.setHeader("Allow", "POST");
    res.setHeader("Access-Control-Allow-Origin", "*");
    res.setHeader("Access-Control-Allow-Methods", "POST");
    res.setHeader("Access-Control-Allow-Headers", "*");
    res.setHeader("Access-Control-Allow-Credentials", "true");

    if (req.method === "OPTIONS") {
      res.status(200).end();
    }
    return await fn(req, res);
  };

const connectDB = (fn: NextApiHandler) =>
  async (req: NextApiRequest, res: NextApiResponse) => {
    await dbConnect();
    return await fn(req, res);
  };

export default connectDB(allowCors(handler));

```

Listing 7-11: The api/graphql.ts file including a connection to the database

We made three changes to the file. First we imported the `dbConnect` function from the middleware; then we created a new wrapper similar to the `allowCors` function and used it to ensure that each API call connects to the API. We could safely do so because we implemented `dbConnect` to enforce only one database connection at the same time. Finally, we wrapped the handler with the new wrapper and exported it as the default.

Adding Services to GraphQL Resolvers

Now it's time to add the services to the resolvers. In Chapter 6, you learned that query resolvers implement the reading of data, whereas mutation resolvers implement the creation, updating, and deletion of data.

There, we also defined two resolvers: one to return a weather object for a given ZIP code and one to update a location's weather data. Now we'll add the services `findByZip` and `updateByZip`, which we created in this chapter,

to the resolvers. Instead of the naïve implementations with the static data object, we modify the resolvers to query and update the MongoDB documents through the services.

Listing 7-12 shows the modified code for the *graphql/resolvers.ts* file in which we refactor these two resolvers.

```
import { WeatherInterface } from "../mongoose/weather/interface";
import { findByZip, updateByZip } from "../mongoose/weather/services";

export const resolvers = {
  Query: {
    weather: async (_, param: WeatherInterface) => {
      let data = await findByZip(param.zip);
      return [data];
    },
  },
  Mutation: {
    weather: async (_, param: { data: WeatherInterface }) => {
      await updateByZip(param.data.zip, param.data);
      let data = await findByZip(param.data.zip);
      return [data];
    },
  },
};
```

Listing 7-12: The graphql/resolvers.ts file using services

We replace the naïve `array.filter` functionality with the appropriate services. To query the data, we use the `findByZip` service and pass it the `zip` variable from the request payload and then return the result data wrapped in an array. For the mutation, we use the `updateByZip` service. Per type definition, the `weather` mutation returns the updated dataset. To do so, we query for the modified document with the `findByZip` service once again and return the result as an array item.

Visit the GraphQL sandbox at <http://localhost:3000/api/graphql> and play with the API endpoints to read and update documents from the MongoDB database.

Summary

In this chapter, you explored using the non-relational database MongoDB and its Mongoose object data modeling tool, which lets you add and enforce schemas as well as perform CRUD operations on MongoDB instances. We covered the differences between relational and non-relational databases and how they store data. Then you created a Mongoose schema and a model, connected Mongoose to the MongoDB instance, and wrote the services to perform operations on the MongoDB collection.

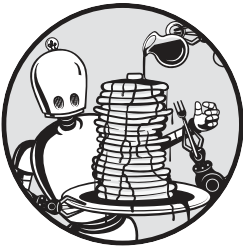
Finally, you connected the REST and GraphQL APIs to the MongoDB database. Now, instead of static datasets, all of your APIs return dynamic documents, and you can both read and update documents through them.

MongoDB and Mongoose are extensive technologies with a huge array of functionalities. To learn more about them, consult the official documentation at <https://mongoosejs.com> and read the articles at <https://www.geeksforgeeks.org/mongoose-module-introduction/>.

The next chapter covers Jest, a modern testing framework for conducting unit, snapshot, and integration tests.

8

TESTING WITH THE JEST FRAMEWORK



Whenever you modify your code, you risk causing unforeseen side effects in another part of your application. As a result, guaranteeing the integrity and stability of a code base can be challenging. To do so, developers follow two main strategies.

In the first, an architectural pattern, we split our code into small, self-contained React components. By nature, these components don't interfere with one another. Hence, changing one shouldn't lead to any side effects. In the second, we perform automated unit testing, which this chapter covers using the Jest framework.

In the following sections, we discuss the essentials of automated unit testing and the benefits of using it. You'll learn how to write a test suite in Jest and use its reports to improve your code. You'll also handle dependencies by using code doubles. Lastly, you'll explore other kinds of tests you might want to run against your application.

Test-Driven Development and Unit Testing

Developers sometimes use the technique of *test-driven development (TDD)*, in which they write their automated tests before implementing the actual code to be tested. They first create a test to evaluate that the smallest possible unit of code would work as expected. Such a test is called a *unit test*. Next, they write the minimum amount of code necessary to pass the test.

This approach has distinct benefits. First, it lets you focus on your app's requirements by explicitly defining the code's functionality and edge cases. Therefore, you have a clear picture of its desired behavior, and you can identify unclear or missing specifications sooner rather than later. When you write tests after completing the functionality, they might reflect the behavior you implemented rather than the behavior you require.

Second, limiting yourself to writing only necessary code prevents your functions from becoming too complex and splits your application into small, understandable sections. Testable code is maintainable code. In addition, the technique ensures that your tests cover a large portion of the app's code, a metric called *code coverage*, and by running the tests frequently during development, you'll instantly recognize bugs introduced by new lines of code.

Depending on the situation, the *unit* targeted by a unit test can be a module, a function, or a line of code. The tests aim to verify that each unit works in isolation. The single lines inside each test function are the test *steps*, and the whole test function is called a test *case*. Test *suites* aggregate test cases into logical blocks. To be considered reproducible, the test must return the same results every time we run it. As we will explore in this chapter, this means that we must run the tests in a controlled environment with a defined dataset.

Facebook developed the Jest testing framework in conjunction with React, but we can use it with any Node.js project. It has a defined syntax for setting up and writing tests. Its *test runner* executes these tests, automatically replaces any dependencies in our code, and generates a test-coverage report. Additional npm modules provide custom code for testing DOM or React components and, of course, adding TypeScript types.

Using Jest

To use Jest in a project, we must install its required packages, create a directory for all test files, and add an npm script that will run the tests. Execute the following in your Next.js application's root directory to install the framework, as well as type definitions from DefinitelyTyped as development dependencies:

```
$ npm install --save-dev jest @types/jest
```

Then create the directory in which to save your tests. Jest uses the `__tests__` folder by default, so make one in your root directory. Next, to add

the npm script `test` to your project, open the `package.json` file and modify the `scripts` object to match the one in Listing 8-1.

```
"scripts": {  
  "dev": "next dev",  
  "build": "next build",  
  "start": "next start",  
  "lint": "next lint",  
  "test": "jest"  
},
```

Listing 8-1: The package.json file with the new test command

Now we can run tests with the `npm test` command. Usually, build servers execute this command by default during the build process. Lastly, to enable TypeScript support in Jest, add the `ts-jest` transpiler:

```
$ npm install --save-dev ts-jest
```

Also create a `jest.config` file to add TypeScript by running `npx ts-jest config:init`.

Creating an Example Module to Test

Let's write some example code to help us understand unit testing and TDD. Say we want to create a new module in our app, `./helpers/sum.ts`. It should export a function, `sum`, that returns the sum of its parameters. To follow a TDD pattern, we'll begin by creating test cases for this module.

First we need to create the function that will run our tests. Create a file called `sum.test.ts` in the default test directory and add the code from Listing 8-2.

```
import { sum } from "../helpers/sum";  
  
describe("the sum function", () => {  
  
});
```

Listing 8-2: The empty test suite

We import the `sum` function we'll write later and use Jest's `describe` function to create an empty test suite. As soon as we run the (nonexistent) tests with `npm test`, Jest should complain that there is no file called `sum.ts` in the `helpers` directory. Create this file and folder now, at the root directory of your project. Within the file, write the `sum` function shown in Listing 8-3.

```
const sum = () => {};  
export { sum };
```

Listing 8-3: The bare bones of the sum function

Now run the tests again with `npm test`. Because the code just exports a placeholder `sum` function that returns nothing, the Jest test runner again complains. This time, it informs us that the test suite needs to contain at least one test.

Let's look at the anatomy of a test case and add a few test cases to the `sum.test.ts` file during the process.

Anatomy of a Test Case

There are two types of unit tests: state and interaction based. An *interaction-based* test case verifies that the code under evaluation invokes a specific function, whereas a *state-based* test case checks the code's return value or resulting state. Both types follow the same three steps: arrange, act, and assert.

Arrange

To write independent and reproducible tests, we need to first *arrange* our environment by defining prerequisites, such as test data. If we need these prerequisites for only one particular test case, we define them at the beginning of the case. Otherwise, we set them globally for all tests in the test suite by using the `beforeEach` hook, which gets executed before each test case, or the `beforeAll` hook, which gets executed before all tests run.

If, for example, we had some reason to use the same global dataset for each test case and knew that our test steps would modify the dataset, we would need to re-create the dataset before each test. The `beforeEach` hook would be the perfect place to do this. On the other hand, if the test cases merely consume the data, we'd need to define the datasets only once and so would use the `beforeAll` hook.

Let's define two test cases and create the input values for each. Our input parameters will be specific to each test case, so we'll declare them inside the test cases instead of using a `beforeEach` or `beforeAll` hook. Update the `sum.test.ts` file with the code from Listing 8-4.

```
import { sum } from "../helpers/sum";

describe("the sum function", () => {
  test("two plus two is four", () => {
    let first = 2;
    let second = 2;
    let expectation = 4;
  });

  test("minus eight plus four is minus four", () => {
    let first = -8;
    let second = 4;
    let expectation = -4;
  });
});
```

Listing 8-4: The test suite containing the arrange steps

The `describe` function creates our test suite, which comprises two calls to the `test` function, each of which is a test case. For both, the first argument is the description we see on the test runner's report.

Each of our tests evaluates the result of the `sum` function. The first checks the addition feature, verifying that 2 plus 2 returns 4. The second test confirms that the function correctly returns negative values as well. It adds 4 to -8 and expects a result of -4 .

You might want to check the return type of the `sum` function, too. Usually, we would have done so, but because we're using TypeScript, there is no need for this additional test case. Instead, we can define the return type in the function signature, and TSC will verify it for us.

Act

As soon as the test runner executes a case, the test steps *act* on our behalf by invoking the code under test with the data for the particular test case. Each test case should test exactly one feature or variant of the system. This step is the line of code that invokes the function to execute. Listing 8-5 adds it to the test cases in `sum.test.ts`.

```
import { sum } from "../helpers/sum";

describe("the sum function", () => {

  test("two plus two is four", () => {
    let first = 2;
    let second = 2;
    let expectation = 4;
    let result = sum(first, second);
  });

  test("minus eight plus four is minus four", () => {
    let first = -8;
    let second = 4;
    let expectation = -4;
    let result = sum(first, second);
  });

});
```

Listing 8-5: The test suite containing the act steps

Our new lines call the `sum` function and pass it the values we defined as parameters. We store the returned values in the `result` variable. In your editor, TSC should throw an error along the lines of Expected 0 arguments, but got 2. This is fine, as the `sum` function is just an empty placeholder and doesn't yet expect any parameters.

Assert

The final step of our test case is the *assertion* that the code fulfills the expectations we defined. We create this assertion with two parts: the Jest `expect`

function, used in conjunction with a *matcher* function from Jest's *assert* library that defines the condition for which we are testing. Depending on the unit test's category, this condition could be a specific return value, a state change, or the invocation of another function. Common matchers check whether a value is a number, a string, and so on. We can also use them to assert that a function returns true or false.

Jest's *assert* library provides us with a built-in set of basic matchers, and we can add additional ones from the npm repository. One of the most common assert packages is *testing-library/dom*, used to query the DOM for a particular node and assert its characteristics. For example, we can check for a class name or attribute or work with native DOM events. Another common assert package, *testing-library/react*, adds utilities for React and gives us access to the render function and React hooks in our asserts.

Because each test case evaluates one condition in one unit of code, we limit each test to one assert. In this way, as soon as the test run succeeds or fails and the test reporter generates the report, we can easily spot which test assumption failed. Listing 8-6 adds one assert per test case. Paste it into the *sum.test.ts* file.

```
import { sum } from "../helpers/sum";

describe("the sum function", () => {

  test("two plus two is four", () => {
    let first = 2;
    let second = 2;
    let expectation = 4;
    let result = sum(first, second);
    expect(result).toBe(expectation);
  });

  test("minus eight plus four is minus four", () => {
    let first = -8;
    let second = 4;
    let expectation = -4;
    let result = sum(first, second);
    expect(result).toBe(expectation);
  });

});
```

Listing 8-6: The test suite containing the assert steps

These lines use the `expect` assert function with the `toBe` matcher to compare the expected result to be the same as our expectation. Our test cases are now complete. Each follows the *arrange, act, assert* pattern and verifies one condition. Appendix C lists additional matchers.

Using TDD

Our test cases still haven't executed, and if you run `npm test`, the test runner should fail immediately. TSC checks the code and throws an error for the missing parameter declarations on the `sum` function:

```
FAIL __tests__/sum.test.ts
  • Test suite failed to run
--snip--
Test Suites: 2 failed, 2 total
Tests:       0 total
Snapshots:   0 total
```

It's time to implement this `sum` function. Following the principles of TDD, we'll incrementally add features to the code and run the test suites after each addition, continuing this process until all tests pass. First we'll add those missing parameters. Replace the code in `sum.ts` with the contents of Listing 8-7.

```
const sum = (a: number, b: number) => {};
```

```
export { sum };
```

Listing 8-7: The `sum` function with added parameters

We've added the parameters and typed them as numbers. Now we rerun the test cases and, as expected, they fail. The console output tells us that the `sum` function doesn't return the expected results. This shouldn't surprise us, because our `sum` function doesn't return anything at all:

```
FAIL __tests__/sum.test.ts (5.151 s)
  the sum function
    × two plus two is four (6 ms)
    × minus eight plus four is minus four (1 ms)

  • the sum function › two plus two is four
    Expected: 4
    Received: undefined

  • the sum function › minus eight plus four is minus four
    Expected: -4
    Received: undefined

Test Suites: 1 failed, 1 total
Tests:       2 failed, 2 total
Snapshots:   0 total
Time:        5.328 s, estimated 11 s
```

The code in Listing 8-8 adds this functionality to the `sum.ts` file. We type the function's return type as a number and add the two parameters.

```
const sum = (a: number, b: number): number => a + b;

export { sum };
```

Listing 8-8: The complete sum function

If we rerun `npm test`, Jest should report that all test cases succeed:

```
PASS  __tests__/sum.test.ts (8.045 s)
  the sum function
    ✓ two plus two is four (2 ms)
    ✓ minus eight plus four is minus four (2 ms)

Test Suites: 1 passed, 1 total
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        8.291 s
```

As you can see, everything worked.

Refactoring Code

Unit tests are particularly useful when we need to refactor our code. As an example, let's rewrite the `sum` function so that, instead of two parameters, it accepts an array of numbers. The function should return the sum of all array items.

We begin by rewriting our existing test cases into a more compact form and then expanding the test suite to verify the new behavior. Replace the code in the `sum.test.file` with Listing 8-9.

```
import { sum } from "../helpers/sum";

describe("the sum function", () => {

  test("two plus two is four", () => {
    expect(sum([2, 2])).toBe(4);
  });

  test("minus eight plus four is minus four", () => {
    expect(sum([-8, 4])).toBe(-4);
  });

  test("two plus two plus minus four is zero", () => {
    expect(sum([2, 2, -4])).toBe(0);
  });

});
```

Listing 8-9: The test suite for the refactored sum function

Notice that we rewrote the test cases in a more compact form. Explicitly splitting the arrange, act, and assert statements across multiple lines may

be easier to read, but for simple test cases, such as those in Listing 8-9, we often write them in one line. We've changed their functionality to accommodate the new requirements. Instead of accepting two values, our `sum` function receives an array with numbers. Again, the TSC instantly notifies us of the mismatching parameters between the `sum` function in the test suite and the actual implementation.

Once we've written our tests, we can rewrite our code. Listing 8-10 shows the code for the `helpers/sum.ts` file. Here the `sum` function now accepts an array of numbers as a parameter and returns a number.

```
const sum = (data: number[]): number => {  
  return data[0] + data[1];  
};  
  
export { sum };
```

Listing 8-10: The rewritten `sum` function in the `helpers/sum.ts` file

We changed the parameter to an array of numbers. This fixes the TypeScript error caused by the test suite in Listing 8-9. But because we're following TDD and making only one functional change at a time, we keep the function's original behavior of adding two values. As expected, one of the test cases fails when we run the automated tests with `npm test`:

```
FAIL  __tests__/sum.test.ts (7.804 s)  
  the sum function  
    ✓ two plus two is four (7 ms)  
    ✓ minus eight plus four is minus four (1 ms)  
    ✗ two plus two plus minus four is zero (9 ms)  
  
    • the sum function › two plus two plus minus four is zero  
      Expected: 0  
      Received: 4  
  
Test Suites: 1 failed, 1 total  
Tests:      1 failed, 2 passed, 3 total  
Snapshots:  0 total  
Time:       8.057 s, estimated 9 s
```

The third test case, which tests the new requirement, is the one that failed. Not only did we expect this result, but we also wanted the test to fail; this way, we know that the tests themselves are working. If they succeeded before we implemented the corresponding functionality, the test cases would be faulty.

With the failing test as the baseline, it is now time to refactor the code to accommodate the new requirement. Paste the code in Listing 8-11 into the `sum.ts` file. Here we refactor the `sum` function to return the sum of all array values.

```
const sum = (data: number[]): number => {
  return data.reduce((a, b) => a + b);
};

export { sum };
```

Listing 8-11: The corrected sum function with array.reduce

Although we could loop through the array with a for loop, we use modern JavaScript's `array.reduce` function. This native array function runs a callback function on each array element. The callback receives the return value of the previous iteration and the current array item as parameters: exactly what we need to calculate the sum.

Run all the test cases in our test suite to verify that they are working as expected:

```
PASS  __tests__/sum.test.ts (7.422 s)
  the sum function
    ✓ two plus two is four (2 ms)
    ✓ minus eight plus four is minus four
    ✓ two plus two plus minus four is zero

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        7.613 s
```

The test runner should show that the code passed every test.

Evaluating Test Coverage

To measure exactly which lines of code our test suites cover, Jest generates a test-coverage report. The higher the percentage of code our tests assess, the more thorough they are, and the more confident we can be about the application's quality and maintainability. As a general rule of thumb, you should aim for code coverage of 90 percent or above, with a high coverage for the most critical part of your code. Of course, the test cases should add value by testing the code's functions; adding tests just to increase the test coverage is not the goal we are aiming for. But as soon as you've tested your code base thoroughly, you can refactor existing features and implement new ones without worrying about introducing regression bugs. A high code coverage verifies that changes have no hidden side effects.

Modify the `npm` test script in the `package.json` file by adding the `--coverage` flag to it, as shown in Listing 8-12.

```
"scripts": {
  "dev": "next dev",
  "build": "next build",
  "start": "next start",
  "lint": "next lint",
```

```
    "test": "jest --coverage"
  },
```

Listing 8-12: Enabling Jest's test-coverage feature in the package.json file

If we rerun the test suite, Jest should show what percentage of the code our unit tests cover. It generates a code-coverage report and stores it in the *coverage* folder. Compare your output with the following:

```
PASS  __tests__/sum.test.ts (7.324 s)
  the sum function
    ✓ two plus two is four (2 ms)
    ✓ minus eight plus four is minus four
    ✓ two plus two plus minus four is zero (1 ms)
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
sum.ts	100	100	100	100	

```
Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        7.687 s, estimated 8 s
```

The report shows the coverage broken down by statements, branches, functions, and lines. We see that our simple `sum` function has a code coverage of 100 percent across all categories. Hence, we know that we've left no code untested and can trust that the test cases reflect the function's quality.

Replacing Dependencies with Fakes, Stubs, and Mocks

We mentioned that our tests should run in isolation, without depending on external code. You might have wondered how to handle imported modules; after all, as soon as you import code, you add a dependency to the unit under evaluation. Those third-party modules might not work as expected, and we don't want our code to depend on the assumption that they all operate correctly. Consequently, you should provide a set of test cases for each imported module to verify its functionality. They, too, are units to test.

Separately, instead of importing modules into our other code units, we need to replace them with *test doubles* that return a defined set of static data tailored to the test. Test doubles replace an object or a function, effectively removing a dependency. Because they return a defined dataset, their response is known and predictable. You can compare them to stunt doubles in movies.

Besides replacing an object or function, test doubles have a second important purpose: they record their calls and let us spy on them. We can thus use them to test whether the test double has been called at all, how often, and which arguments it received. There are three main types of test doubles: fakes, stubs, and mocks. However, you'll sometimes hear the term *mock* to refer to all three.

Creating a Module with Dependencies

To practice using test doubles in our `sum` function, we'll create a new function that calculates a specified number of values in the Fibonacci sequence. The *Fibonacci sequence* is a pattern in which each subsequent number is the sum of the previous two, a simple use case for a `sum` module.

All developers must figure out how fine-grained their test cases need to be. The Fibonacci sequence is a good example, because trying to test every possible number submitted to the function would be useless, as the sequence has no end. Instead, we want to verify that the function properly handles edge cases and that its underlying functionality works. For instance, we'll check how it handles an input with a length of 0; in that case, the function should return an empty string. Then we'll test how it calculates a Fibonacci sequence of any length longer than 3. Create the *fibonacci.test.ts* test suite inside the `__tests__` folder and then add the code from Listing 8-13 to it.

```
import { fibonacci } from "../helpers/fibonacci";

describe("the fibonacci sequence", () => {

  test("with a length of 0 is ", () => {
    expect(fibonacci(0)).toBe("");
  });

  test("with a length of 5 is '0, 1, 1, 2, 3' ", () => {
    expect(fibonacci(5)).toBe("0, 1, 1, 2, 3");
  });

});
```

Listing 8-13: The test suite for the *fibonacci* function

We define two test cases: one that checks for a length input of 0 and another that calculates a Fibonacci sequence of five numbers. Both tests follow the *arrange, act, assert* pattern in the compact variant we used earlier.

After we've created the test cases, we can move on to writing the Fibonacci function code. Create the *fibonacci.ts* file in the *helpers* folder, next to the *sum.ts* file, and add the code from Listing 8-14 to it.

```
import { sum } from "./sum";

const fibonacci = (length: number): string => {
  const sequence: number[] = [];
  for (let i = 0; i < length; i++) {
    if (i < 2) {
      sequence.push(sum([0, i]));
    } else {
      sequence.push(sum([sequence[i - 1], sequence[i - 2]]));
    }
  }
}
```

```
    return sequence.join(", ");  
  };  
  
export { fibonacci };
```

Listing 8-14: The fibonacci function

We import the `sum` function from the module we created earlier in this chapter. It is now a dependency that we'll need to replace with a test double later. Then we implement the `fibonacci` function, which accepts the length of the sequence to calculate and returns a string. We store the current sequence in an array so that we have a simple way to access the two previous values needed to calculate the next one. Notice that the first number in the sequence is always 0 and the second is always 1. Finally, we return a string with the requested number of values. If you save this code and rerun the test suites, both `sum.test.js` and `fibonacci.test.ts` should pass successfully.

Creating a Doubles Folder

Because we import the `sum` function in the `Fibonacci` module, our code has an external dependency. This is problematic for testing purposes: if the `sum` function breaks, the test for the Fibonacci sequence will fail as well, even if the logic of the Fibonacci implementation is correct.

To decouple the test from the dependency, we'll replace the `sum` function in the `fibonacci.ts` file with a test double. Jest can replace any module that has an identically named file saved in a `__mocks__` subdirectory adjacent to the test file during the test run. Create such a folder in the `helpers` folder next to the test file and place a `sum.ts` file inside it. Leave the file empty for now.

To enable the test double, we call the `jest.mock` function, passing it the path to the original module saved in the test file. In Listing 8-15, we add this call to `fibonacci.test.ts`.

```
import { fibonacci } from "../helpers/fibonacci";  
  
jest.mock("../helpers/sum");  
  
describe("the fibonacci sequence", () => {  
  test("with a length of 0 is ", () => {  
    expect(fibonacci(0)).toBe("");  
  });  
  
  test("with a length of 5 is '0, 1, 1, 2, 3' ", () => {  
    expect(fibonacci(5)).toBe("0, 1, 1, 2, 3");  
  });  
});
```

Listing 8-15: The test suite for the fibonacci function with the test double

This new line replaces the `sum` module with the test double. Now let's create all three basic types of test doubles, adding their code to the file in the `__mocks__` folder.

Using a Stub

Stubs are merely objects that return some predefined data. This makes them very simple to implement but limited in use; often, returning the same data isn't enough to mimic a dependency's original actions. Listing 8-16 shows a stub implementation for the `sum` function's test double. Paste the code into the `sum.ts` file inside the `__mocks__` folder.

```
const sum = (data: number[]): number => 999;

export { sum };
```

Listing 8-16: A stub for the `sum` function

The stubbed function has the same signature as the original function. It accepts the same arguments, an array of numbers, and returns a string. Unlike the original, however, this test double always returns the same number, 999, regardless of the data it received.

To successfully run the test suites with this stub function, we'd need to adjust our expectations about what our code will do. Instead of returning five numbers in the Fibonacci sequence, it would produce the string 999, 999, 999, 999, 999. If we see such a string, we know that the `sum` function was called five times. Experiment with the stub, modifying the test suite's expectations to match it. Then restore the matchers to those shown in Listing 8-15 so that you can use them for the upcoming tests.

Using a Fake

Fakes are the most complex kind of test double. They are working implementations of the original functionality, but unlike the real implementation, they provide only the functionality necessary for the unit test. Their implementation is simplified and often doesn't cater to edge cases.

The fake for the `sum` adds the first and second items of the array manually, instead of using `array.reduce`. This simplified implementation strips the `sum` function of the ability to sum more than two data points, but for the Fibonacci sequence, it is sufficient. The reduced complexity makes it easy to understand and less prone to error. Replace the content of the `sum.ts` file inside the `__mocks__` folder with the code in Listing 8-17.

```
const sum = (data: number[]): number => {
  return data[0] + data[1];
}

export { sum };
```

Listing 8-17: A fake for the `sum` function

Our fake uses a simple mathematical plus operator (+) to add the first and second items of the `data` parameter. Its main benefit is that it returns a result similar to that of the actual implementation. We can now run the test

suites, and they should pass successfully without our having to adjust our expectations, returning the Fibonacci sequence.

Using a Mock

Mocks lie somewhere between stubs and fakes. Although less sophisticated than fakes, they return more realistic data than stubs. While they don't simulate a dependency's true behavior, they can react to the data they receive.

For example, our naive mock implementation of the `sum` function will return a result from a hardcoded hash map. Replace the code in the `__mocks__/sum.ts` file with the code from Listing 8-18, which inspects the request and enables the Fibonacci calculator to use the original test suites.

```
type resultMap = {
  [key: string]: number;
}

const results : resultMap = {
  "0+0": 0,
  "0+1": 1,
  "1+0": 1,
  "1+1": 2,
  "2+1": 3
};

const sum = (data: number[]): number => {
  return results[data.join("+")];
}

export { sum };
```

Listing 8-18: A mock for the `sum` function

We create a type, called `resultMap`, that uses a string as a key and a number as a value. Then we use the newly created type for a hash map that stores our desired responses. Next, we define the mock function with the same interface as the original implementation. Inside the mock, we calculate the key to use in the hash map based on the parameters we receive. This lets us return the correct dataset and produce an actual Fibonacci sequence. The main benefit of using the mock over `sum` is that we can control its outcome, as it returns values from a known dataset.

Conveniently, Jest provides us with helpers to work with test doubles. The `jest.mock` function replaces imported modules with mocks. The `jest.fn` API creates a basic mock that can return any kind of predefined data, and `jest.spyOn` lets us record calls to any function without modifying it. We will use all of those in Exercise 8 on page 146.

In typical developer contexts, you won't bother with the subtle differences between stubs, fakes, and mocks and will use the term *mock* as a generic term for test doubles. Don't spend too much time overengineering your mocks; they're just tools to help you test your code.

Additional Types of Tests

The tests covered in this chapter so far are the most common ones you'll encounter as a full-stack developer. This section briefly explains additional types of tests and when to use them. These aren't meant to replace unit tests; rather, they supplement unit tests by covering specific aspects of your implementation that aren't otherwise testable. For example, because unit tests run in isolation, they can't evaluate the interaction between modules. Theoretically, if every function and module passes its test, the whole program should work as expected. Practically, you'll often face issues caused by faulty module documentation. Commonly, the documentation will claim that an API returns a specific type, but the actual implementation will return a different one.

Functional Tests

While unit tests examine the implementation of a feature from a developer's perspective, *functional tests* cover the user's perspective by verifying that code works as the user expects it to work. Put otherwise, these tests check that a given input results in an expected output. Most functional tests are a type of *black-box* test, which ignores the module's internal code, side effects, and intermediate results and tests only the interfaces. Functional tests do not generate a code-coverage report. Generally, a quality assurance manager will write and use functional tests during a system testing stage. By contrast, developers write and use unit tests during development.

Integration Tests

You learned that the goal of a unit test is to check the smallest possible section of code in isolation. An *integration test* is the complete opposite. It verifies the behavior of entire subsystems, whether they be layers of code, such as an app's data storage mechanism, or specific functions consisting of multiple modules. Integration tests check the integration of the subsystem in the context of the current environment. Hence, they never run in isolation and typically don't use test doubles.

Integration tests are helpful for finding three types of problems. The first type is problems related to *inter-module communication*, which is the communication between modules. Common problems are faulty internal API integrations and undetected side effects, such as a function that doesn't delete old files before writing new data to the filesystem. The second type is problems related to the *environment*, which describes the hardware and software setup the code runs on. Different software versions or hardware configurations can introduce significant issues for your code. The most common problem for full-stack developers involves differences in Node.js versions and outdated dependencies in the modules.

The third type is problems related to *gateway communications*, which relates to testing any API communication with a third-party API gateway. Any communication with external APIs should be tested with integration tests. This is the only instance in which integration tests might use test

doubles, such as stubbed versions of the external API, in order to simulate a specific API behavior, like a timeout or successful request. As with functional tests, a quality assurance manager generally writes and uses integration tests. Developers do so less often.

End-to-End Tests

You can think of the *end-to-end test* as a combination of functional tests and integration tests. As another kind of black-box test, they examine the application's functionality across the full stack, from the frontend to the back-end, in a specific environment. These business-facing tests should provide confidence that the overall application is still working as expected.

End-to-end tests run the application in a specific environment. Often, the complexity of the many dependencies increases the risk of flaky tests in which the application works correctly but the environment causes the tests to fail. End-to-end tests are thus the most time-consuming to create and maintain. Due to their complexity, we must craft them carefully. During execution, they are known to be slow, prone to encountering timeouts, and, like nearly all black-box tests, unable to provide detailed error reports. Therefore, they test only the most critical business-facing scenarios. Generally, a quality assurance manager writes them.

Snapshot Tests

The tests described earlier in this chapter check the code against some assertion. By contrast, a *snapshot test* compares the application's current visual (or user interface) state to a previous version of it. Hence, these tests are also called visual regression tests. In each test, we create new snapshots and then compare them with ones stored earlier, providing a cheap way to test user interface components and complete pages. Instead of manually creating and maintaining tests that describe every property of an interface, such as a component's height, width, position, and colors, we can establish a snapshot containing all of these properties.

One way to perform this type of test is to create and compare screenshots. Generally, a headless browser renders the component; the test runner waits for the page to render and then captures an image of it. Unfortunately, this process is relatively slow, and headless browsers are flaky. Jest takes a different approach to snapshot testing. Instead of working with headless browsers and image files, it renders the React user interface components to the virtual DOM, serializes them, and saves them as plain-text in *snap* files stored in the `__snapshots__` directory. Hence, Jest snapshot tests are much more performant and less flawed. The Food Finder application you'll build in Part II will use snapshot tests to verify the integrity of the build and test your React components.

Exercise 8: Add Test Cases to the Weather App

As long as you follow the basic principles we've discussed, there is no right or wrong way to test your code. Unit, snapshot, and end-to-end tests are all different tools in your tool belt, and you must strike a balance between the time you spend writing the tests and the usefulness of each. There is also no consensus on what to test. While you should strive for 90 percent code coverage or higher, the general rule of thumb is to cover at least the most critical parts of your application with unit tests and then write some integration tests to verify that your application works on each deployment.

When it comes to our weather application, we'll want our test cases to cover four core aspects. First we'll add unit tests to evaluate the middleware and services. Even though the REST API endpoints and React user interface component are easy to test directly in the browser, we'll add test cases for both of them: a basic snapshot test for the user interface component and an end-to-end test for the REST API endpoint `/v1/weather/[zipcode].ts`.

We've opted to test the REST endpoint rather than the GraphQL API for simplicity's sake, as each REST endpoint has its own file, while all GraphQL APIs share an entry point, making their testing more complex. However, testing this GraphQL API would make an excellent exercise for exploring end-to-end-tests after you've finished the chapter.

Testing the Middleware with Spies

The middleware to connect to the database is a core part of the application, but we can't access it directly, as it doesn't expose any API. We can only implicitly test it by examining the database or by running a query through Mongoose, some service, or an API endpoint. Each of these methods would work, but if we want to test the connection to the database as a unit test, we need to do so in a way that isolates that component as much as possible.

To do so, we'll use Jest's built-in spies to verify that our middleware successfully calls all the functions necessary for establishing the connection to the MongoDB memory server. Navigate to your `__tests__` folder and create a new folder, `middleware`, and a file, `db-connect.test.ts`, inside it. Then copy the code from Listing 8-19 into the file.

```
/**
 * @jest-environment node
 */

import dbConnect from "../../middleware/db-connect";
import mongoose from "mongoose";
import { MongoMemoryServer } from "mongodb-memory-server";

describe("dbConnect ", () => {

  let connection: any;

  afterEach(async () => {
    jest.clearAllMocks();
```

```

        await connection.stop();
        await mongoose.disconnect();
    });

    afterAll(async () => {
        jest.restoreAllMocks();
    });

    test("calls MongoMemoryServer.create()", async () => {
        const spy = jest.spyOn(MongoMemoryServer, "create");
        connection = await dbConnect();
        expect(spy).toHaveBeenCalled();
    });

    test("calls mongoose.disconnect()", async () => {
        const spy = jest.spyOn(mongoose, "disconnect");
        connection = await dbConnect();
        expect(spy).toHaveBeenCalled();
    });

    test("calls mongoose.connect()", async () => {
        const spy = jest.spyOn(mongoose, "connect");
        connection = await dbConnect();
        const MONGO_URI = connection.getUri();
        expect(spy).toHaveBeenCalledWith(MONGO_URI, {dbName: "Weather"});
    });
});

```

Listing 8-19: The `__tests__/middleware/db-connect.test.ts` suite for the database connection

Most of this code resembles the test suites you wrote earlier in this chapter. But instead of testing simplified example code, we're now testing real code, which requires us to make some adjustments.

First we set the testing environment for Jest to `node`, which simulates a Node.js runtime. Later, when writing snapshot tests, we'll use Jest's default environment, called `jsdom`, which simulates a browser by providing a `window` object, as well as all the usual DOM properties and functions. By always setting these environments in the file, we avoid issues caused by using the wrong environment. Then, as usual, we import the packages we need.

Now we can start writing the test suite for the `dbConnect` function. We define a `connection` variable in the test suite's scope to store the database connection, and then we can access the MongoDB's server instance, including its methods and properties. For example, we'll use these to stop the connection and disconnect from the server after each test to guarantee that each test case is independent.

To be able to store the connection, we first need to return the `mongoServer` constant from the `dbConnect` function in the file `db-connect.ts`. Open the file and add the line `return mongoServer` to the `dbConnect` function right before the function's closing curly bracket `}`. From time to time, you'll need to modify

the code you wrote earlier to accommodate the requirements of your tests. In other words, you need to adapt the code to make it testable.

Now we use the connection we just exposed and set up the `afterEach` hook, which runs after each test case, to reset the mocked functions to their initial mocked state, thus clearing all previously gathered data. This is necessary, because otherwise, the spies would report information gained during the previous calls, as they retain their state across all test suites. Also, we re-create the database connection for each test case. Therefore, we need to stop the current connection and explicitly disconnect from the database after each test. Then we set up the `afterAll` hook to remove all mocks and restore the original functions through the `restoreAllMocks` function.

Our test cases should all follow the *arrange, act, assert* pattern. As we review them, you might find it useful to open the `db-connect.ts` file in the `middleware` folder and follow along. The initial test case verifies the call to the `create` function on the `MongoMemoryServer`, as this is the first function that we call in the `db-connect.ts` file. To do so, we create a spy with the `jest.spyOn` method. As arguments, this method takes the name of an object and the object's method on which to spy. Then we act on the code under test and call the `dbConnect` function. Finally, we assert that the spy has been called.

The second test case works similarly except that it spies on a different method. We use it to check that `mongoose.disconnect` was called successfully during the execution of `dbConnect`. The third test case introduces a new matcher. Instead of verifying only the call itself with `toHaveBeenCalled`, we now also verify the call's arguments using `toHaveBeenCalledWith`. Here we grab the connection string directly from the connection and store it in the variable `MONGO_URI`. We also hardcode the database we want to connect to. Then we call the matcher, passing it the expected arguments and verifying that they meet our expectations.

Now run the test suites with `npm test`. All tests should pass with 100 percent test coverage.

Creating Mocks to Test the Services

While the tests we wrote for the middleware were quite simple, the service tests are a bit more complicated. If you open the `mongoose/weather/services.ts` file, you'll see that the services depend on `WeatherModel`, which is Mongoose's gateway to the MongoDB collection. Each service calls a method on the model that, in turn, requires a database connection. We won't reevaluate those database connections here; instead, the goal of this test suite will be to verify that the service functions call the correct `WeatherModel` functions. To do so, we'll create a mock `WeatherModel` that exposes the same set of APIs as mocked functions.

We first write the mocked model. Following convention, we create the file `mongoose/weather/__mocks__/model.ts` and add the code in Listing 8-20.

```
import { WeatherInterface } from "../interface";

type param = {
  [key: string]: string;
};

const WeatherModel = {
  create: jest.fn((newData: WeatherInterface) => Promise.resolve(true)),
  findOne: jest.fn(({ zip: paramZip }: param) => Promise.resolve(true)),
  updateOne: jest.fn(({ zip: paramZip }: param, newData: WeatherInterface) =>
    Promise.resolve(true)
  ),
  deleteOne: jest.fn(({ zip: paramZip }: param) => Promise.resolve(true))
};
export default WeatherModel;
```

Listing 8-20: The mock for the WeatherModel

We implement `WeatherInterface` and define the new `param` type, which is an object with key-value pairs that we use to type the first parameter. We make the mocked `WeatherModel` the default export and use an object that implements the four methods of the actual `WeatherModel`, each of which takes the same parameters as the original. They also take the original Mongoose model's method. Because they are asynchronous functions, we return a promise resolved to `true`.

Now we can write the test suite for the services. These check that each service returns `true` upon success and calls the correct method of the mocked `WeatherModel`. Create the file `/__tests__/mongoose/weather/services.test.ts` and add the code from Listing 8-21 to it.

```
/**
 * @jest-environment node
 */
import { WeatherInterface } from "../../mongoose/weather/interface";
import {
  findByZip,
  storeDocument,
  updateByZip,
  deleteByZip,
} from "../../mongoose/weather/services";

import WeatherModel from "../../mongoose/weather/model";
jest.mock("../../mongoose/weather/model");

describe("the weather services", () => {

  let doc: WeatherInterface = {
    zip: "test",
    weather: "weather",
    tempC: "00",
    tempF: "01",
    friends: []
  };
});
```

```

afterEach(async () => {
  jest.clearAllMocks();
});

afterAll(async () => {
  jest.restoreAllMocks();
});

describe("API storeDocument", () => {
  test("returns true", async () => {
    const result = await storeDocument(doc);
    expect(result).toBeTruthy();
  });
  test("passes the document to Model.create()", async () => {
    const spy = jest.spyOn(WeatherModel, "create");
    await storeDocument(doc);
    expect(spy).toHaveBeenCalledWith(doc);
  });
});

describe("API findByZip", () => {
  test("returns true", async () => {
    const result = await findByZip(doc.zip);
    expect(result).toBeTruthy();
  });
  test("passes the zip code to Model.findOne()", async () => {
    const spy = jest.spyOn(WeatherModel, "findOne");
    await findByZip(doc.zip);
    expect(spy).toHaveBeenCalledWith({ zip: doc.zip });
  });
});

describe("API updateByZip", () => {
  test("returns true", async () => {
    const result = await updateByZip(doc.zip, doc);
    expect(result).toBeTruthy();
  });
  test("passes the zip code and the new data to Model.updateOne()", async () => {
    const spy = jest.spyOn(WeatherModel, "updateOne");
    const result = await updateByZip(doc.zip, doc);
    expect(spy).toHaveBeenCalledWith({ zip: doc.zip }, doc);
  });
});

describe("API deleteByZip", () => {
  test("returns true", async () => {
    const result = await deleteByZip(doc.zip);
    expect(result).toBeTruthy();
  });
  test("passes the zip code Model.deleteOne()", async () => {
    const spy = jest.spyOn(WeatherModel, "deleteOne");
    const result = await deleteByZip(doc.zip);
  });
});

```



```

        expect(spy).toHaveBeenCalledWith({ zip: doc.zip });
    });
});
});

```

Listing 8-21: The updated test suite in `__tests__/mongoose/weather/services.test.ts`

As in the previous test suite, we begin by setting up the environment and importing modules. We also import `WeatherModel` and call `jest.mock` with the path to the mocked model we created, effectively replacing the original model in the code under test. Then we create a document containing some test data. We store it in the constant `doc` and will pass it to the mocked model's methods. As done previously, we use the `afterEach` hook to reset all mocks after each test and the `afterAll` hook to remove the mocks and restore the original functions after all test cases have been finished.

We create a nested test suite for each of the four services. Each has the same two unit tests: one to verify the return value upon success with the `toBeTruthy` matcher and one to spy on a particular `WeatherModel` mock function. The code follows the same pattern as the previous test suite and uses the same matchers as well.

The code-coverage report we receive after running `npm test` shows that we tested around 70 percent of the service code. If you take a look at the uncovered lines listed in the last column, you'll see that they contain the `console.log(err);` output. This output is used whenever an asynchronous call to the model's methods fails:

PASS	__tests__/mongoose/weather/services.test.ts				
PASS	__tests__/middleware/dbconnect.test.ts (7.193 s)				
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	83.63	100	88.23	82.35	
middleware	100	100	100	100	
db-connect.test.ts	100	100	100	100	
mongoose/weather.	77.41	100	100	75.86	
services.test.ts	70.83	100	100	70.83	8,20-22,33-35,43-45

For the purposes of this chapter, we'll leave these lines uncovered. Otherwise, we could modify the mocked model to throw an error—for example, by supplying an invalid document—and then add a third test case per service verifying the error.

Performing an End-to-End Test of the REST API

Sophisticated API tests might use a dedicated API testing library such as *SuperTest*, which provides matchers for HTTP status codes and simplifies the handling of requests and responses. Alternatively, they might use a GUI

tool like Postman. In this example, we'll merely test that the returned data matches our expectations by using the native `fetch` method.

Unlike the previous tests, this one doesn't isolate any single component, as our goal is to verify that all components of the system work together as expected. To check whether the API returns a proper response from the database when supplied some input, our end-to-end test will make certain assumptions: that all layers have already been tested independently, that the database contains its initial seed data, and that our application runs at `http://localhost:3000/`.

To verify our first assumption, open the API endpoint file `pages/api/v1/weather/[zipcode].ts`. You'll notice that the API code imports two functions, `findByZip` from the service module and the middleware's `dbConnect`, both of which we've already tested. The second assumption is also satisfied; the database loads the initial seed on each startup. Create the file `zipcode.e2e.test.ts` in `__tests__/pages/api/v1/weather/` and add the code from Listing 8-22.

```
/**
 * @jest-environment node
 */

describe("The API /v1/weather/[zipcode]", () => {
  test("returns the correct data for the zipcode 96815", async () => {
    const zip = "96815";
    let response = await fetch(`http://localhost:3000/api/v1/weather/${zip}`);
    let body = await response.json();
    expect(body.zip).toEqual(zip);
  });
});

export {};
```

Listing 8-22: The test suite for the REST API

We set the environment to `node` and then define the test suite with one test case. In it, we supply a ZIP code that matches one of the initial seed datasets. Then we use the native `fetch` method, which has been available since Node.js version 17.5, to call our weather API on our localhost and check whether the returned ZIP code is the same as the one passed as a parameter. We add an empty `export` statement to define this file as an ES6 module.

The test should pass and have 100 percent code coverage. Now that we're confident that the core of our application is working as expected, we can test the user interface components.

When using `fetch`, there are two common error messages you might encounter. The first, `ECONNREFUSED`, tells you that `fetch` could not connect to your application because it is not running. Use `npm run dev` to start the application or adjust the port in the `fetch` call if you're not using port 3000. The second error mentions that the test exceeded the timeout of 5,000 ms for a test. If you started your application for the purposes of testing and did not use a previously running application, Next.js compiles the API route as soon as the test consumes it. Depending on your environment, this might take longer than the default timeout. Add the line `jest.setTimeout(20000);`

before the `describe` method at the top of your file to increase the timeout and make the test wait 20,000 ms instead of 5,000 ms.

Evaluating the User Interface with a Snapshot Test

Snapshot tests verify that a page's rendered HTML didn't change between two test runs. To achieve this with Jest, we must first prepare our environment. Add the `jsdom` environment, `react-testing-library`, and the `react-test-renderer` to the project:

```
$ npm install --save-dev jest-environment-jsdom
$ npm install --save-dev @testing-library/react @testing-library/jest-dom
$ npm install --save-dev @types/react-test-renderer react-test-renderer
```

We'll need these to simulate a browser environment and render React components during our test cases. Now we'll modify the `jest.config.js` file in our root directory accordingly. Replace its content with the code in Listing 8-23.

```
const nextJest = require("next/jest");
const createJestConfig = nextJest({});

module.exports = createJestConfig(nextJest({}));
```

Listing 8-23: The updated `jest.config.js` file

This code imports the `next/jest` package and exports a Jest configuration with the default properties of a Next.js project. It is the simplest form of Next.js-compatible Jest configuration. If you take a look at the official Next.js setup guide at <https://nextjs.org/docs/testing>, you'll see that it outlines some basic configuration options, none of which we need.

The First Version

A snapshot test renders a component or a page, takes a snapshot of it as serialized JSON, and stores it in a `__snapshots__` folder next to the test suite. On each consecutive run, Jest compares the current snapshot with the stored reference. As long as they are the same, the snapshot test passes. To generate the initial snapshot, create a new folder, `__tests__/pages/components`, and the file `weather.snapshot.test.tsx`, and then add the code in Listing 8-24 to it.

```
/**
 * @jest-environment node
 */

import { act, create } from "react-test-renderer";
import PageComponentWeather from "../../pages/components/weather";

describe("PageComponentWeather", () => {
  test("renders correctly", async () => {
    let component: any;
    await act(async () => {
      component =
    
```

```

        await create(<PageComponentWeather></PageComponentWeather>);
      });
      expect(component.toJSON()).toMatchSnapshot();
    });
  });
});

```

Listing 8-24: The snapshot test for PageComponentWeather

The first lines of our snapshot test set the environment to `jsdom` and import the test renderer's `act` and `create` methods to test the React component, which we import in the next line.

Next, we write the simulated user behavior and wrap the component's creation in the asynchronous `act` function. As you might have guessed, this function draws its name from the *arrange, act, assert* pattern and ensures that all relevant updates to the DOM have been applied before proceeding with the test case. It is required for all statements that cause updates to the React state, and here, it delays the test execution until after the `useEffect` hook runs.

Then we write a test case that awaits the `create` function, which renders the JSX component. This lets us generate HTML in a simulated browser environment and store the result in a variable. We await the component's rendering so that the HTML is available for our follow-up interactions before we continue with the test case. Then we serialize the rendered component to a JSON string and use a new matcher, `toMatchSnapshot`, which compares the current JSON string with the stored reference.

A trial run shows that all tests succeed. We see two interesting things—that the test created one snapshot and that we achieved a test coverage of 81 percent:

```

PASS  __tests__/mongoose/weather/services.test.ts
PASS  __tests__/pages/api/v1/weather/zipcode.e2e.test.ts
PASS  __tests__/middleware/dbconnect.test.ts (7.193 s)
PASS  __tests__/pages/components/weather.snapshot.test.tsx

```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
All files	83.63	100	88.23	82.35	
middleware	100	100	100	100	
db-connect.test.ts	100	100	100	100	
mongoose/weather	77.41	100	100	75.86	
services.test.ts	70.83	100	100	70.83	8,20-22,33-35,43-45
pages/api/v1/					
weather					
[zipcode].ts	100	100	100	100	
pages/components	81.81	100	60	80	
weather.tsx	81.81	100	60	80	8,12

```

Snapshot Summary
  › 1 snapshot written from 1 test suite.

```

You can look at the created snapshot by opening the *weather.snapshot.test.tsx.snap* file in the `__snapshots__` folder. It should look fairly similar to the code in Listing 8-25, and you'll see that it is nothing more than the rendered HTML saved as a multiline template literal. Your HTML might not be identical to that shown here; the important aspect is that it looks the same after each test run when `react-test-renderer` rendered the component.

```
// Jest Snapshot v1, https://goo.gl/fbAQLP

exports[`PageComponentWeather renders correctly 1`] = `
<h1
  data-testid="h1"
  onClick={[Function]}
>
  The weather is
  sunny
  , and the counter shows
  0
</h1>
`;
```

Listing 8-25: The `weather.snapshot.test.tsx.snap` file with the serialized HTML

We also see that the counter is set to 0, which indicates that the `useEffect` did not run before we created the snapshot. If you open the component's file and check the uncovered lines, you'll learn that they relate to the click handler that increases the state variable and, as suspected, the `useEffect` hook. We want to test these core functionalities as well.

The Second Version

We'll modify the test code to cover the previously untested functionalities. Paste the code from Listing 8-26 into the snapshot test file.

```
/**
 * @jest-environment node
 */

import { act, create } from "react-test-renderer";
import PageComponentWeather from "../../pages/components/weather";

describe("PageComponentWeather", () => {
  test("renders correctly", async () => {
    let component: any;
    await act(async () => {
      component = await create(<PageComponentWeather></PageComponentWeather>);
    });
    expect(component.toJSON()).toMatchSnapshot();
  });

  test("clicks the h1 element and updates the state", async () => {
    let component: any;
```

```
    await act(async () => {
      component = await create(<PageComponentWeather></PageComponentWeather>);
      component.root.findByType("h1").props.onClick();
    });
    expect(component.toJSON()).toMatchSnapshot();
  });
});
```

Listing 8-26: The updated snapshot test

In the updated code, we've added another test case that finds the headline on the page and simulates a user clicking it. Remember from previous chapters that this increases the state variable counter. Again, we await the creation of the component and use the `act` function.

If you rerun the tests, you should see a failure. The test runner tells us that the snapshots do not match:

```
FAIL __tests__/pages/components/weather.snapshot.test.tsx
  • PageComponentWeather › renders correctly
--snip--
  › 1 snapshot failed.
--snip--
Snapshot Summary
  › 1 snapshot failed from 1 test suite.
  › Inspect your code changes or run `npm test -- -u` to update them.
```

Because we modified the test case to wait for the `useEffect` hook and set the state variable counter to 1 instead of 0, the DOM changed as well. Follow the test runner's advice and rerun the tests with `npm test -- -u` to create a new, updated snapshot. The tests should now succeed, reporting a test coverage of 100 percent for our component.

Try experimenting with your newfound knowledge. For example, can you write a snapshot test for the page routes in the *pages* directory or a set of end-to-end tests for the GraphQL API?

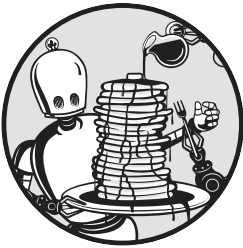
Summary

You should now be able to create automated tests with Jest and, more broadly, design a testing plan on your own to strike a balance between effort and reward. We discussed the benefits of TDD and unit testing and then used the *arrange, act, assert* pattern to develop a simple `sum` function following test-driven principles. Next, we used three types of test doubles to replace the `sum` function while calculating the Fibonacci sequence. Finally, we added unit and snapshot tests to our existing Next.js application, created a mock of a Mongoose model, and used spies to verify our assumptions.

To learn more about Jest and automated testing, consult the official Jest documentation at <https://jestjs.io/docs/getting-started>. In the next chapter, you'll explore the differences between authorization and authentication and how you can leverage OAuth in your applications.

9

AUTHORIZATION WITH OAUTH



Certain apps store data about users as part of a login workflow. There are many ways to implement this authentication and authorization, but one of the easiest is to use OAuth2 to piggyback on the existing accounts of well-known companies. *OAuth2*, or simply *OAuth*, is an open standard for access delegation, and you’ve probably encountered it if you’ve ever used an app’s “log in with Facebook, GitHub, or Google Account” feature.

The OAuth protocol essentially allows our web application to access another application’s login data without requiring the third party to share a user’s credentials with us. To do so, the user grants our application access rights to their third-party account through the creation of an access token. OAuth is the accepted standard for authorization-based access delegation, and Amazon, Google, Facebook, Microsoft, and GitHub all support OAuth workflows.

This chapter will introduce you to the OAuth workflow and then explore the structure of the bearer tokens used for its access delegation, laying the foundation for implementing OAuth2 into your Food Finder application in Part II. In Exercise 9 on page 168, we won't update our sample Next.js application with an OAuth flow but instead manually walk through the OAuth authorization process.

How OAuth Works

Before we explore OAuth, you need to understand the differences between authentication and authorization. In short, we use authentication to verify the identity of a user, whereas authorization specifies the permissions that the authenticated user possesses and enforces those permissions. OAuth allows for that process to be delegated to a third party with which the user already has an account, which simplifies the login process for the user.

Authentication vs. Authorization

Every time an app receives a login request, it checks the user's credentials before allowing access, a process called *authentication*. Usually, those credentials consist of a username and a password, but they could also be hardware tokens or involve biometric factors such as fingerprints or facial recognition. The application then verifies that the credentials match the ones stored in the database.

The simplest form of authentication is *single-factor authentication*, which requires only one factor, usually a password. Unfortunately, it is also the least secure method of implementing authentication. A more robust and recommended form is *multifactor authentication*, in which a user must supply at least two factors. These might be something the user *knows*, such as a password, as well as something the user *has*, such as a physical token, or something the user *is*, such as the owner of a fingerprint. You probably use multifactor authentication when you log in to PayPal or Google, both of which require you to supply your password and an additional one-time password (OTP).

The OTP is a code that is created based on a secret shared between you and the application when you register your account. Both actors regenerate the pair in short intervals. Yours may be generated by an authenticator app, like Google Authenticator, or received in a text message. The application at which you have the account (for example, PayPal or Google) generates its own OTP code and keeps it on the server. As soon as you send yours, the server verifies that the codes cryptographically match.

We perform authorization after we've authenticated a user. Broadly speaking, this involves looking at the user's data and deciding whether they have the access rights needed to access a resource. A typical full-stack application can either handle this user data or enable users to log in without providing user data. There are benefits to the latter approach, as handling and storing user data can be inconvenient. It also comes with additional

responsibilities, such as the need to adhere to stricter privacy and data retention laws, and requires your users to create another account.

Suppose you provide users with the option to log in with an existing account through an authorization provider. In that case, you've removed an entry barrier. Also, you don't need to worry about handling their data. If you need user data—for example, to bill your customers—you can use an OAuth workflow and save the data you receive from the provider, such as the user's payment details, in your own database if necessary.

The Role of OAuth

Every time a web application enables you to log in through a third-party provider such as Facebook, GitHub, or Google, it uses the OAuth authorization code flow behind the scenes. OAuth isn't authentication; rather, it's a way of authorizing the web application you use to perform actions or to access resources on your behalf. Common actions include posting to your Facebook feed and accessing data such as your name, profile picture, or email address. Consequently, each time you use an OAuth-based login function, the application asks for particular permissions and can use only those you grant to it.

To understand OAuth, you must understand its terminology. Each OAuth flow uses a set of RESTful APIs to authorize the *client* (an application) to get resources (such as the user's profile information) from a *resource provider* (such as Facebook, GitHub, or Google) that has the protected resources the client wants to access. In addition, we call the server that provides the OAuth API endpoints the *authorization server*, and the party that owns the access rights (and, hence, has the ability to grant an application access to a resource) the *resource owner*. In most scenarios, the resource owner is the application's end user.

To get the resource owner's authorization, the client application sends its client credentials, the ID, the secret, and the user credentials to the authorization server, which usually is part of the same system as the resource provider. The authorization server authenticates the resource owner and handles the OAuth flow that results in granting them an *access token*, which allows the user to access the protected resources on the resource provider. Both the authorization server and the resource provider are two sets of APIs on the same system.

The *client ID* is a public identifier for the client app; you can make it public and store it in the code. Unlike the client ID, the *client secret* should be kept private; it is the app-specific password, and you should never store it in your code. Instead, handle it using Next.js's environment files or your server's environment variables.

Grant Types

There are several variants of the OAuth flow. Each of these *grant types* covers a specific use case, but all result in the generation of an access token. OAuth specifies four grant types: the client credentials flow, the implicit

flow, the authorization code flow, and the resource owner password credentials flow.

The *client credentials flow* covers machine-to-machine communication; we use it when no actual end user authorization is necessary, as in the case of automated tasks that connect to an API. Here, the task itself is both the client and the resource owner. It knows the resource owner's credentials, the client ID, and the client secret and passes these to the authorization server to receive an access token.

The most common grant type for full-stack web development is the *authorization code flow*. In this scenario, our web application is a client, and it makes two calls to two separate API endpoints. The first is to receive an authorization grant code, and the second is to exchange this authorization grant for an access token. “The Authorization Code Flow” on page 161 provides a deep dive into this process.

The last two grant types shouldn't be used. The *implicit flow* is similar to the authorization code flow, but instead of making separate requests to receive the authorization grant and access token, the client receives the access token directly. This flow skips the authorization step, doesn't include client authentication, and is deprecated. The *resource owner password credentials flow* should be avoided because it involves the end user passing their user credentials to the client and then the client sending these credentials to the OAuth server to exchange them for the access token. While this sounds straightforward, sending actual user credentials to the remote authorization server is an immense security risk.

Bearer Tokens

After the client application initiates an OAuth flow, it receives a shared access token, most commonly a *bearer token* that is easy to implement. This access token replaces the user's credentials; hence, anyone who has the token can access the data. To prevent security gaps caused by stolen tokens, a bearer token usually has a defined shelf life. Upon expiration, the token can be refreshed only with a valid *refresh token*. These are long-lived tokens that we use to generate new bearer tokens.

Refreshing the token can be done implicitly or explicitly, and there are multiple strategies for preventing stolen refresh tokens from compromising the OAuth access. For example, the OAuth provider can require a unique ID or the client secret to issue a new token. The provider usually rotates the refresh token each time a new bearer token is issued and accepts each refresh token only once. From our perspective as OAuth clients, the details of the refresh token are unimportant, as the OAuth provider handles this token.

The bearer token that contains the user session and authentication data is a *JSON Web Token (JWT)*. JWT is an open standard for securely transmitting data in a JSON object. Because JSON is fairly compact, JWTs can be sent as URL parameters, as part of the POST data, or even inside an HTTP header, all without impacting the application's performance.

JWT tokens can be signed as well as encrypted, saving the application from needing to make an additional request to verify it or retrieve extra

data. *Encrypted tokens* hide the contained data from other parties. These aren't very common in OAuth due to their additional overhead, so we can ignore them for now. *Signed tokens* guarantee the integrity of the contained data, because any modification to the token would change its signature. Thus, the application can trust the information stored in it.

The most common cryptographic algorithm for signing JWTs is *hash-based message authentication code (HMAC)* with the SHA-256 hash algorithm. An HMAC is a type of message authentication code (MAC). A MAC's main feature is that it enables you to verify the authenticity of a message by calculating a checksum from the message. The checksum uses a mathematical function to produce a unique, reproducible value or data string based on the initial message. If the message changes, the checksum changes as well. This way, we can quickly verify the integrity of the data. For the JWT token, we use two checks: the authenticity check confirms that the actual sender sent the message, whereas the data integrity checks verify that the message's content did not change.

Unlike other types of MACs, HMAC uses a cryptographic hash function and a secret key. You can freely choose the cryptographic hash function, but the strength of your HMAC implementation depends on the cryptographic strength of the selected function. JWTs commonly use the SHA-256 hash function, a fast and collision-resistant cryptographic function from the SHA-2 collection also used for authenticating Debian software packages and Bitcoin transactions. In cryptography, *collisions* occur when two different inputs result in the same output. The higher the possibility of a collision, the less we can trust the checksum of the hash function. If a collision is likely, our message could be replaced with a different one, but the hash function could indicate that it hasn't changed. Therefore, we want collision-resistant cryptographic functions.

The Authorization Code Flow

To understand how an OAuth interaction takes place using the authorization code flow mentioned earlier, let's return to our fictional weather service. Imagine that you want to grant weather stations the ability to write data to the application by using the API, but a station should be able to modify only its own ZIP code. You also want the application to display the weather stations' locations and additional details about them. Additionally, you prefer not to deal with the maintenance of user accounts or to manually set up permissions for each station, so using OAuth is your best bet.

Let's assume that each weather station already has a social media account for publishing weather updates. These accounts include typical user information and the stations' ZIP codes. We could easily use the social media provider as an OAuth authorization provider to access this data. The stations would log in to the weather app using the social media provider, and the app would request access to the weather station's user profile. We could then check the ZIP code stored in the OAuth session against the one in our dataset, provide the appropriate write access, and retrieve any other data we need.

Only a few steps are necessary for implementing this authorization code flow. Figure 9-1 is a simplified description of these steps. Usually, developers use an SDK or a Node.js module to implement the steps and need to provide only a few properties, such as the client ID, client secret, and callback URL.

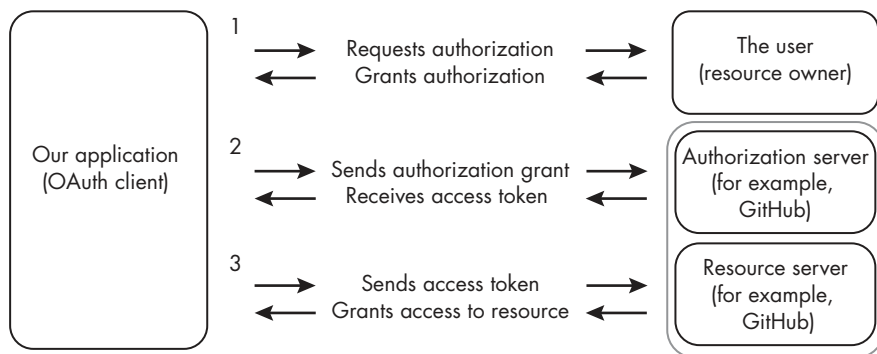


Figure 9-1: A simplified OAuth authorization grant flow

To register our app as an OAuth client, we need to provide GitHub with a *callback URL* to our application, to which GitHub will redirect the user after the authorization request. This endpoint on our application receives the authorization grant. Recent OAuth implementations require the callback URL to use HTTPS as a way to protect the token from being intercepted.

Our app must use the resource owner’s credentials and the client credentials, an ID, and a secret to communicate with GitHub’s authorization server. The ID identifies the client, and the secret authenticates it. The app can then request the authorization to access specific resources, such as a weather station’s profile data. To do so, the weather station user needs to log in to GitHub’s authorization server. They’ll see a prompt that summarizes the requested access resources, such as read and write access to the profile or stream. If the user authorizes the requests with their user credentials, the OAuth client receives the authorization grant as a GET parameter in the callback URL, and the OAuth SDK we use in our application exchanges the authorization grant for an access token at the authorization server in the next step of the flow.

Here, the OAuth client uses the client credentials, which are the client ID and client secret, in combination with the previously received authorization grant to request an access token from the OAuth provider’s authorization server. It is part of the GitHub infrastructure, and to complete the authorization flow, the authorization server authenticates the identity and verifies that the grant is valid for this identity. Finally, the app receives the bearer token from here and stores it in the user session.

With the token and the user session received from the OAuth provider, our app can now act on the user’s behalf and access their protected

resources, such as the profile data from the resource server. To act on their behalf, we add the bearer token to the Authorization header in the HTTP requests; the OAuth provider checks our granted permission and verifies our identity with this token. To access the user's data, we simply extract it from the session data and use it in our application's code.

For the weather application, we could use the second option to query location-specific weather data from our database. We'd need to read the location property from the user's session data and use that value as the ZIP code supplied to our API endpoint. In addition, we can access other properties, such as the description and the name or profile picture, to display them on the weather application's status page for each station.

Creating a JWT Token

Most bearer tokens are JWTs, and while the authorization server automatically issues them, it's good to know what kind of information you can find in them. This section will walk you through the process of creating an example OAuth JWT for the weather service app. The JWT is a string made up of three sections divided by periods (.): the header, the payload, and the signature. The first two sections are Base64-encoded JSON objects, whereas the signature is a checksum of the previous two.

The Header

The first string we create is the *header*, which defines basic metadata such as the token's type and the signatures used for the signing algorithm. Listing 9-1 shows the creation of a simple header in JavaScript with the most essential metadata.

```
const headerObject = {  
  "typ": "JWT",  
  "alg": "HS256"  
}
```

Listing 9-1: The JWT header for the OAuth2 weather service

We set the type of the weather service's token to JWT and specify that we use the HMAC-SHA-256 algorithm to calculate the signature later. Finally, we store the JSON object in a constant to use later.

The Payload

Next, we create the second string, the *payload*, which stores the token's data. Each property of the payload is called a *claim*. In OAuth, the claims describe the user object and, usually, the session data. The JWT specification contains three types of claims: registered, public, and private.

Registered Claims

There are seven registered claims, each three letters long. While not necessary in general JWTs, the `iss`, `sub`, `auth`, and `exp` registered claims are required for OAuth JWTs.

The *issuer claim*, `iss`, contains a unique identifier for the entity that issued the JWT. A good value might be the application's URL, as shown in Listing 9-2.

```
{
  "iss": "https://www.usemodernfullstack.dev/"
}
```

Listing 9-2: A registered issuer claim

The *subject claim*, `sub`, identifies the principal to which the JWT belongs. For an OAuth client authentication flow, the subject claim must be the client ID of the OAuth client, whereas for an OAuth authorization grant, the subject should identify the resource owner or should pseudonymously identify an anonymous user. We create a sample subject claim in Listing 9-3.

```
{
  "sub": "THE_CLIENT_ID"
}
```

Listing 9-3: A registered subject claim

The *audience claim*, `aud`, identifies the token's recipient. Its value could be the token endpoint URL on the authorization server or anything else that identifies the recipient, such as an application ID. See Listing 9-4 for an example.

```
{
  "aud": "api://endpoint"
}
```

Listing 9-4: A registered audience claim

The *expiration claim*, `exp`, identifies the time window during which the token is valid. After that period, the authorization server will reject the token and you'll need to request a new one. An expiration claim's value is a number whose date is defined in "seconds since the Unix Epoch," a common format for timestamps. It is calculated by counting the number of seconds that have elapsed since January 1, 1970. Listing 9-5 shows an example.

```
{
  "exp": 1134156400
}
```

Listing 9-5: A registered expiration claim

The *issued at claim*, *iat*, is optional and identifies the time at which the authorization server issued the token. You can determine a token's age from this claim, which is also defined in seconds since the Unix Epoch, as shown in Listing 9-6.

```
{  
  "iat": 1134156200  
}
```

Listing 9-6: A registered issued at claim

The *not before claim*, *nfb*, is optional and identifies the time at which the authorization server should start accepting the token. The authorization server will reject every token with an *nfb* claim in the future. We define it as a number in seconds since the Unix Epoch, as you can see in Listing 9-7.

```
{  
  "nfb": 1134156100  
}
```

Listing 9-7: A registered not before claim

The *JWT claim*, *jti*, is optional and sets a unique ID for the token (see Listing 9-8).

```
{  
  "jti": "b5f8f86f-82ab-451e-b391-bf6a07041787"  
}
```

Listing 9-8: A registered JWT claim

The authorization server might keep a list of recent tokens and their expiration dates to check whether the token is being reused in a *replay attack*, which occurs when an attacker tries to access data by reusing a previously issued token.

Public Claims

A token's issuer can define public claims for the purpose of adding an application-specific public API. Unlike private claims, these are custom properties defined for public access. The issuer should register these claims in the JWT Claims registry or use collision-resistant names with custom namespaces—for example, a UUID or the application's name. Also, as public claims are meant for public consumption, they should never include private or sensitive information.

A public claim for the OAuth JWT of our fictional weather service might include the ZIP code to directly provide each station's location data. By making the ZIP code a public claim, we won't need to parse the user object and extract the ZIP code manually. Also, as the location is publicly available information on social media profiles, it's not sensitive.

Private Claims

Private claims are custom claims that are neither registered claims nor public claims. We can define them to our liking, and they can be specific to our application or use case. Even though they don't need to be collision resistant, using a private namespace is recommended. Unlike public claims, private claims contain information specific to the application and are intended to be used only internally. Whereas the public claims store generic information such as the name, the private claims contain the application's user ID and role. For example, we could define a private claim for the OAuth JWT of our fictional weather service to specify the type of service we are using.

Now that you understand the payload object's possible properties, you can create a complete payload like the one in Listing 9-9, which specifies GitHub as the service.

```
const payloadObject = {
  "exp": 234133423,
  "weather_public_zip": "96815",
  "weather_private_type": "GitHub"
}
```

Listing 9-9: The JWT payload for the OAuth weather service

Again, we create a constant and store the object there. Our payload has three claims, each of a different type. It's up to the publisher of the JWT token to decide which claims to include; for this example, we limit the size of the token to one of each type. The registered claim `exp` sets the expiration date and time, `zip` is a public claim, and `role` is a private claim. Both use the custom namespace `weather` to minimize the risk of a collision.

The Signature

With the header and payload in place, we create a JWT signature by using the algorithm specified in the header to calculate the checksum. We pass the header and payload as Base64-encoded strings and a custom secret to the checksum function. As an exercise, we'll create the signature in TypeScript with the code from Listing 9-10. You'll see that the secret is hardcoded for simplicity here. In production code, this secret should be stored in an environment variable.

Save the code as `index.ts` in a TypeScript project, or use `npx ts-node index.ts` to run it locally. If you prefer, you can also use a TypeScript sandbox at <https://codesandbox.io> or <https://stackblitz.com> to run it. Generate a fresh secret (<https://www.usemodernfullstack.dev/generate-secret>) and use it instead of the one in the listing to see how the token changes.

```
import { createHmac } from "crypto";

const base64UrlEncode = (data: string): string => {
  return Buffer.from(data, "utf-8").toString("base64");
};
```

```

const headerObject = {
  typ: "JWT",
  alg: "HS256"
};

const payloadObject = {
  exp: 234133423,
  weather_public_zip: "96815",
  weather_private_type: "GitHub"
};

const createJWT = () => {
  const base64Header = base64UrlEncode(JSON.stringify(headerObject));
  const base64Payload = base64UrlEncode(JSON.stringify(payloadObject));

  const secret = "59c4b48eac7e9ac37c046ba88964870d";

  const signature: string = createHmac("sha256", secret)
    .update(`${base64Header}.${base64Payload}`)
    .digest("hex");

  return [base64Header, base64Payload, signature].join(".");
};

console.log(createJWT());

```

Listing 9-10: An index.ts file to calculate the JWT signature for the OAuth2 weather service

We use Node.js's standard crypto module and then create a library for transforming the JSON objects into Base64-encoded strings via buffers. We pass the strings and the secret to the crypto module's createHmac function to initialize the HMAC object with sha256 as the hashing algorithm. Then we feed the Base64-encoded header and payload string, separated by a period, to the HMAC object. Finally, we convert the result to a hexadecimal format.

To obtain the JWT string, we create an array containing the Base64-encoded strings from the header and payload objects, as well as the Base64-encoded signature. To convert the array into a string that uses a period to separate every part, we call `Array.join` with a period as a separator and return the resulting JWT.

To generate the JWT, we run the script. The final JWT token logged to the console should look similar to the one in Listing 9-11.

```

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOiJzNDZMzQyMywid2VhdGhlc19wdWJsaWNfemlwIjo0IjY4MTUzZWF0aGVyX3ByaXZhdGVfdHlwZSI6IkdpdEh1YiJ9.f667c81749886e01831376a38fbdba4d7f59a14c14f3a60e1bbee977c993ac9

```

Listing 9-11: The final JWT token for the OAuth2 weather service

In the next section, we'll use our new knowledge to walk through an actual OAuth flow.

Exercise 9: Access a Protected Resource

Now that you understand OAuth's components and the theory behind the authorization code flow, let's work with a practical example. We'll try to access the protected resource hosted by an OAuth server at <https://www.usemodernfullstack.dev/protected/resource>. Run the exercise's cURL commands from your terminal to follow along.

First, attempt to access the protected resource without an access token by sending a GET request for it:

```
$ curl -i \
  -X GET 'https://www.usemodernfullstack.dev/protected/resource' \
  -H 'Accept: text/html'
--snip--
HTTP/2 401
Content-Type: text/html; charset=utf-8
--snip--
<h1>Unauthorized request: no authentication given</h1>
```

We use the `-i` flag to output the headers, and when we search the response for the HTTP code, we see a `401` status code, which tells us that we're not authorized to access the resource and must obtain an access token.

To get an access token, we'll set up an OAuth client by creating a user account and registering it with the provider to receive a client ID and client secret. Then we'll make a request to the `/oauth/authorize` endpoint, log in with the user's credentials, and receive the authorization grant on our callback URL. Next, we'll exchange the grant code for an access token on the `/oauth/access_token` endpoint. Finally, we'll make the same request again, providing the access token in the header.

The callback URL can be any URL here, as we're not sending any actual data to it. But for a real authorization grant flow, it needs to be an endpoint on your application. Usually, an OAuth SDK provides these, as it handles the response and tokens.

Setting Up the Client

Before we start the OAuth flow, we need to create a user and register an OAuth client. Open <https://www.usemodernfullstack.dev/register> in your browser. On the form shown in Figure 9-2, create a user account with a username and password of your choice.

www.usemodernfullstack.dev/register

MODERN FULL STACK DEVELOPMENT PORTAL & OAUTH SERVER

First Name *
Martin

Last Name *
Krause

Username *
martinkrause

Password *
example

Confirm Password *
example

Email *
martinkrause@example.com

☒ By registering I agree to the Privacy Policy *

Create an Account and move on to the OAuth Client

Create only your Account

Figure 9-2: Creating a user account with the OAuth provider

Then proceed to register a client by providing a callback URL (Figure 9-3). This callback URL points to the OAuth callback endpoint on our application. Usually, the SDK or the OAuth provider supplies you with instructions on how to set this up.

www.usemodernfullstack.dev/register#

MODERN FULL STACK DEVELOPMENT PORTAL & OAUTH SERVER

Create a user and client at
usemodernfullstack.dev

Then, register a new OAuth client at the OAuth provider
usemodernfullstack.dev
Write down the Client ID and the Client Secret

Client ID *
client-1682049053929

Client Secret *
b2d951cff56e6039f6be017acef6a57d

Redirect URIs *
http://localhost:3000/oauth/callback

Register your OAuth Client

Figure 9-3: Registering a client application with the OAuth server to receive the client credentials

The form is prefilled with a callback URL similar to a typical OAuth callback structure. Usually, you find them in the SDK's documentation. Don't worry that the URL `http://localhost:3000/oauth/callback` doesn't exist on your application. For this exercise, we won't send any actual data to it; instead, we'll see that it's part of the request and response flow when we go through the API calls. Click the button to move on to the next step, where you create the OAuth client. Make sure to write down your username, password, client ID, and client secret. You'll need all of these for the next steps. Then click **Register Your OAuth Client** to complete the process.

Logging In to Receive the Authorization Grant

Now the user we registered must use their credentials to log in to the OAuth provider, allowing the client application to access their resources. We call the OAuth REST API endpoint `/oauth/authorize` and (as the resource owner) log in with our user credentials, which is the first step of the flow. The API response returns a redirect to the callback URL, which contains the authorization grant in the URL parameter code.

In a real application, the resource owner would click some "Log in with OAuth" button and enter their credentials, and the API calls would happen behind the scenes. But for the purposes of this exercise, we'll perform all API requests manually. By using the raw API calls, we'll see the actions that SDKs usually abstract. Call the REST endpoint directly with the following cURL command:

```
$ curl -i \
  -X POST 'https://www.usemodernfullstack.dev/oauth/authenticate' \
  -H 'Accept: text/html' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d "response_type=code\
&client_id=<OAUTH_CLIENT_ID>\
&state=4nBjkh31\
&scope=read\
&redirect_uri=http://localhost:3000/oauth/callback\
&username=<OAUTH_USER>\
&password=<OAUTH_PASSWORD>"
--snip--
HTTP/2 302
Content-Type: text/html; charset=utf-8
location: http://localhost:3000/oauth/callback?code=<AUTHORIZATION_GRANT>&state=4nBjkh31
```

This POST request logs in to the OAuth provider. We set the URL to the `oauth/authenticate` endpoint, as well as our Accept header and the appropriate Content-Type header, `application/x-www-form-urlencoded`, for form data.

We use the `-d` flag to send the POST data indicating that we're looking for an authorization code. To split the POST data into readable chunks, we need to use double quotes (") to wrap it and the backslash (\) for line breaks. We add the client ID we received from the OAuth provider and the callback URL we discussed earlier. The scope parameter specifies the permissions we're asking for, while the state parameter

contains a unique random string that mitigates cross-site request forgery (CSRF) attacks. The OAuth provider should return this state parameter along with the authorization code so that we can verify that its value hasn't changed, proving that the response originated from the correct API and not from a third party. In addition, we send the user credentials we registered before.

The response headers show us that everything worked as expected. The OAuth API responds with a status code of *302* and redirects to the callback URL we provided. As you can see in the location header, the redirect to the callback URL contains the authorization grant in the code parameter, as well as the state parameter we sent. Unlike the state, which is just being reflected, the authorization grant is unique and depends on the request data.

Using the Authorization Grant to Get the Access Token

Next, we use the authorization grant to request an access token from the OAuth server. Copy the code you received in the preceding step and use it to request the bearer access token with the client credentials from the */oauth/access_token* API endpoint:

```
$ curl -i \
  -X POST 'https://www.usemodernfullstack.dev/oauth/access_token' \
  -H 'Accept: text/html, application/json' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d "code=<AUTHORIZATION_GRANT>\
    &grant_type=authorization_code\
    &redirect_uri=http://localhost:3000/oauth/callback\
    &client_id=<OAUTH_CLIENT_ID>\
    &client_secret=<OAUTH_CLIENT_SECRET>"
--snip--
HTTP/2 200 OK
Content-Type: application/json; charset=utf-8

{
  "access_token": "9bd55e2acf046128a54b76eada1ea6e0f909ca53",
  "token_type": "Bearer",
  "expires_in": 3599,
  "refresh_token": "79a22d2b37c635a6095f5548ca08ea632deae573",
  "scope": "read"
}
```

This POST request to the OAuth server uses the Accept header to accept a JSON response and sets the Content-Type header to a value for POST form data. We send the form data itself with the data-raw flag. The data contains the authorization grant we received in the code parameter, a grant_type parameter that tells the API endpoint to expect an authorization grant flow, and the same redirect URL as before. We also pass in the client ID and secret.

The response has an HTTP status code of *200*, which means the request succeeded. In the response body, we received the access token and additional details. Copy the access token's value for the next step.

Using the Access Token to Get the Protected Resource

We now have an access token from the OAuth server that we can use to retrieve the protected resource we couldn't access at the beginning of this exercise. Use the same `cURL` command to request `https://www.usemodernfullstack.dev/protected/resource`, and replace the `ACCESS_TOKEN` placeholder with the access token:

```
$ curl -i \
  -X GET 'https://www.usemodernfullstack.dev/protected/resource' \
  -H 'Accept: text/html' \
  -H 'Authorization: Bearer <ACCESS_TOKEN>'
--snip--
HTTP/2 200 OK
Content-Type: text/html; charset=utf-8
--snip--
<h1>This page is secured.</h1>
--snip--
```

We use the `Authorization` header with the `Bearer` keyword and the access token we received from the authorization grant flow in the `access_token` property. When we search for the HTTP status code, we see that instead of a code of `401`, we receive a code of `200`. On closer inspection, we also see that the response's body contains the secured content.

We manually walked through all the necessary steps for receiving a working access token. This exercise is appropriate for educational purposes only; as mentioned earlier in this chapter, we usually use an SDK or a library such as *next-auth* to implement an OAuth flow.

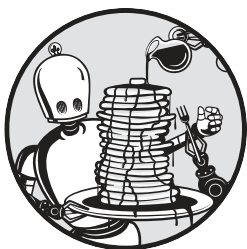
Summary

Authentication involves using credentials to authorize access, whereas authorization defines and grants access rights. This chapter covered implementing authorization with the OAuth2 protocol. You became familiar with the authorization grant flow, the most common OAuth flow used in full-stack web applications, and learned how to create JWTs. Then you practiced manually working with OAuth, getting and using the JWT bearer token, and applying the OAuth flow to your application from a bird's-eye view.

You can find additional resources, tutorials, and specifications at <https://oauth.net>. The next chapter covers Docker, a containerization platform that decouples your development environment from your local machine.

10

CONTAINERIZATION WITH DOCKER



Professional full-stack developers frequently work with Docker and, more broadly, containers. *Docker*, an open source containerization platform, solves three common problems.

First, it lets us run a particular version of some software, such as Node.js, for each of our projects. Second, it decouples the development environment from our local machine and creates a reproducible way to run the application. Third, unlike traditional virtual machines, Docker containers run on a shared host. Therefore, they are smaller in size and consume less memory than classic virtual machines, which emulate a complete system and are often hardware specific. As a result, container-based applications are lightweight and easy to scale. These advantages have made Docker the most appreciated development platform in recent years.

This chapter covers the fundamentals of Docker. We first walk through the steps required to containerize our Next.js application by creating a Docker container running the latest Node.js version and serving the application from inside the container. Then we explore the concept of a microservice architecture and create two microservices using Docker.

The Containerization Architecture

In their daily lives, developers must regularly switch between applications that require different versions of the same library. For example, a JavaScript-focused developer might need a different Node.js or TypeScript version for each of their projects. Of course, they could switch the installed Node.js version on their local machine with tools such as `nvm` whenever they need to work on a different project. But instead of resorting to crude hacks, they could choose a more elegant solution.

Using Docker, we can separate our application or its services into independent containers, each of which provides a service-specific environment. These containers run on an operating system of our choosing (often Debian, Ubuntu, or Alpine), with only the dependencies necessary to this particular application. Containers are isolated from one another and communicate through defined APIs.

When we use a Docker container during the development process, we facilitate the application's later deployment. After all, the container provides a location-independent version of our application that is platform agnostic. Therefore, we already know that our application works with the installed dependencies and that no conflicts or additional installation steps are necessary. Instead of setting up a remote server with the required software and then deploying and testing our application afterward, we can simply move our Docker container to the server and spin it up there.

In situations when we need to move to a different server, scale our application, add additional database servers, or distribute instances across several locations, Docker lets us deploy our application by using the same straightforward process. Instead of managing different hosts and configurations, we can effectively build a platform-agnostic application and run the same containers everywhere.

Installing Docker

To check whether you already have Docker installed, open the command line and run `docker -v`. If you see a version number higher than 20, you should be able to follow along with the examples in this chapter. Otherwise, you'll need to install the most recent version of Docker from Docker Inc. Go to <https://www.docker.com/products/docker-desktop/>. Then choose the Docker desktop installer for your operating system and download it. Execute the application and check the Docker version number on the command line. It should match the one you downloaded.

Creating a Docker Container

Docker has several components. The physical or virtual machine on which the Docker daemon runs is the *host system*. While you're developing your

application locally, the host is your physical machine, and when you deploy your container, the host is the server that runs the application.

We use the *Docker daemon service* on the host system to interact with all components of the Docker platform. The daemon provides Docker's functionality through APIs and is the actual Docker application installed on our machine. Access the daemon using the `docker` command from the command line. Run `docker --help` to display all possible interactions.

We use Docker *containers* to run our containerized applications. These containers are running instances of a particular Docker image, which is the artifact that contains the application. Each Docker image relies on a Dockerfile, which defines the configuration and the content of the Docker image.

Writing the Dockerfile

A *Dockerfile* is a text file containing the information we need to set up a Docker image. It commonly builds upon some existing base image, such as a bare-bones Linux machine on which we've installed additional software or a pre-provisioned environment. For example, we might use a Linux image with Node.js, MongoDB, and all relevant dependencies installed.

Often, we can build upon an official image. For example, Listing 10-1 shows the basic Dockerfile we use to containerize our refactored Next.js application. Dockerfiles contain keywords followed by commands, and we use the `FROM` keyword here to select the official Node.js Docker image. Create a file called *Dockerfile* in your project's root directory, next to the *package.json* file, and add the code in Listing 10-1 to it.

```
FROM node:current

WORKDIR /home/node
COPY package.json package-lock.json /home/node/
EXPOSE 3000
```

Listing 10-1: A simple Dockerfile for a typical Node.js-based application

The image we've selected contains a preconfigured Node.js system running on Debian. The version tag `current` gives you the most recent Node.js version; alternatively, we could provide a particular version number here. Hence, if you need to lock any application to a specific Node.js version, this is the line to do so. You could also use the slimmer `node:current-slim` image, a lightweight Debian distribution that contains only the software packages necessary to run Node.js. However, we need MongoDB's in-memory server, so we'll choose the regular image. You can see a list of the available images at <https://hub.docker.com>. Other images you'll probably use in your career include those for WordPress, MySQL, Redis, Apache, and NGINX.

Finally, we use the `WORKDIR` keyword to set the working directory inside the Docker image to the user's home directory. All future commands will

now execute in this directory. We use the `COPY` keyword to add the *package.json* and *package-lock.json* files to the working directory. A Node.js application runs on port 3000 by default, so we use the `EXPORT` keyword to choose port 3000 for TCP connections. This connection will provide access to the application from outside the container.

Building the Docker Image

To create a Docker image from the Dockerfile, we use the `docker image build` command. During the build process, the Docker daemon reads the Dockerfile and executes the commands defined there to download and install software, copy local files into the image, and configure the environment. Run the following next to your Dockerfile to build the image from it:

```
$ docker image build --tag nextjs:latest .
[+] Building 11.9s (10/10) FINISHED
=> [internal] load build definition from Dockerfile                                0.1s
=> => transferring dockerfile: 136B                                              0.0s
=> [1/2] FROM docker.io/library/node:current-alpine@sha256:HASH 0.0s
=> [2/2] WORKDIR /home/node                                                    0.0s
=> => naming to docker.io/library/ nextjs:latest
```

The `--tag` flag gives the image the name `nextjs` and sets its version to `latest`. Now we can easily refer to this specific image at a later time. We use a period (`.`) at the end of the command to set the build context, limiting the `docker build` command's file access to the current directory. In the output, the Docker daemon indicates that it successfully built the tagged image.

Now, to verify that we have access to the image, run the following. This command lists all locally available Docker images:

```
$ docker image ls
REPOSITORY    TAG       IMAGE
nextjs        latest    98b28358e19a
```

As expected, our newly created image has a random ID (`98b28358e19a`), is tagged as `nextjs`, and is available in the `latest` version. The Docker daemon may also display additional information, such as the size and age of the image, which aren't relevant to us for now.

Docker provides additional commands for managing local and remote images. You can view a list of all available commands by running `docker image --help`. For example, to remove an existing image from your local machine, use `docker image rm`:

```
$ docker image rm <name:version or ID>
```

After a while, you'll find that you've collected unused or outdated versions of your images, so deleting them to free up space on your machine with `docker image prune` is a good practice.

Serving the Application from the Docker Container

Docker containers are running instances of Docker images. You could use the same Docker image to spin up multiple containers, each with a unique name or ID. Once the container is running, you can synchronize local files to it. It listens on an exposed TCP or UDP port, so you can connect to it and execute commands inside it using SSH.

Let's containerize our application. We'll spin up the Docker container from our image, map the local Next.js files to the working directory, publish the exposed port, and finally start the Next.js development server. We can do all of this using `docker container run`:

```
$ docker container run \
--name nextjs_container \
--volume ~/nextjs_refactored/:/home/node/ \
--publish-all \
nextjs:latest npm run dev
> refactored-app@0.1.0 dev
> next dev
```

```
ready - started server on 0.0.0.0:3000, url: http://localhost:3000
event - compiled client and server successfully in 10.9s (208 modules)
```

At first glance, this command might look complicated, but once we take a closer look at it, you'll easily understand what it is doing. We pass it several flags, starting with the `--name` flag, which assigns a unique name to the running container. We'll use this name to identify the container later.

Then we use the `--volume` flag to create a Docker volume. *Volumes* are a simple way to share data between containers. Docker itself manages them, and they let us synchronize our application files to the `home/node/` directory inside the container. We use the format *source:destination* to define a volume, and depending on your file structure, you might need to adjust the absolute path to this folder. In this example, we map `/nextjs_refactored/` from the user's home folder into the container.

The `--publish-all` flag publishes all exported ports and assigns them to random ports on the host system. We use `docker container ls` later to view the ports for our application. The last two arguments are intuitive: `nextjs:latest` points to the Docker image we want to use for the container, and `npm run dev` starts the Next.js development server as usual. The console output shows that the Node.js app inside the container is running and listening on port 3000.

Locating the Exposed Docker Port

Unfortunately, as soon as we try to access our Next.js application on port 3000, the browser notifies us that it isn't accessible; no application is listening there. The problem is that we didn't map the exposed Docker port 3000 to the host's port 3000. Instead, we used the `--publish-all` flag and assigned a random port to the exposed Docker port.

Let's run `docker container ls` to see details about all running Docker containers:

```
$ docker container ls
```

CONTAINER ID	IMAGE	PORTS	NAMES
dff681898013	nextjs:latest	0.0.0.0:55000->3000/tcp	nextjs_container

Search for the name we assigned to our container, *nextjs_container*, and notice that port 55000 on the host maps to the Docker port 3000. Hence, we can access our application at *http://localhost:55000*. Open this URL in your browser. You should see the Next.js application.

If you glance at the URL bar, you'll notice that the port we use to access the application is different from the one used in previous chapters because it is now running inside the Docker container. Try to access all of the pages and APIs we created previously before moving to the next section.

Interacting with the Container

You can view a list of all Docker commands for interacting with containers by running `docker container --help`. In most contexts, though, you'll find it sufficient to know just a few of these. For example, use `exec` to execute commands inside an already running Docker container. We could use `exec` to connect to a shell inside the container by passing it the `-it` flag and the path to the shell, such as `/bin/sh`. The `-i` flag is short for `--interactive`, whereas `-t` runs a pseudoterminal. The interactive option lets us interact with the container, and the `tty` pseudoterminal keeps the Docker container running so that we can actually interact with it:

```
$ docker container exec -it <container ID or name> /bin/sh
```

The `kill` command stops a running Docker container:

```
$ docker container kill <containerid or name>
```

We can select the container by name or by using the container ID shown in the list of local running containers.

Creating Microservices with Docker Compose

Docker provides us with a way to break up an application into small, autonomous units, called *microservices*. A microservice-driven architecture splits an application into a collection of self-contained services that communicate through well-defined APIs. It's a relatively new architectural concept that gained traction around the late 2000s to early 2010s, when Docker and other tools that allowed for easier partitioning and orchestration of server resources became available. These tools form the technical foundation of a microservice architecture.

Microservices have several advantages. First, each independent service has a single purpose, which reduces its complexity. Therefore, it is more testable

and maintainable. We can also deploy the microservices separately, spin up multiple instances of a single microservice to improve its performance, or swap it out altogether without affecting the whole application. Contrast these features with a traditional monolithic application whose user interface, middleware, and data storage exist in one single program built from a single code base. Even if a monolith uses a more modular approach, the code base couples them tightly, and you can't swap out the elements easily.

Another characteristic feature of microservices is that dedicated teams can own just a single service and its code base. This means that they can select the appropriate tools, frameworks, and programming languages on a per-service basis. On the other hand, you'd typically use a single core language to write a monolithic application.

Now that you know how to create a single container from scratch, we'll practice creating multiple containers; each will serve one part of an application. One way to use microservices is to create one service for the frontend and a second for the backend. The Food Finder application we'll create in Part II will use this structure. The main benefit of this approach is that it lets us use a preconfigured MongoDB image for the database. For the example in this chapter, we'll create a second service that watches our weather service and reruns its test suite as soon as the file changes. To do so, we'll use the Docker Compose interface and define our microservice architecture in a *docker-compose.yml* file.

Writing the *docker-compose.yml* File

We define all services in *docker-compose.yml*, a text file in the YAML format. This file also sets the properties, dependencies, and volumes for each service. Most properties are similar to the command line flags you specify when creating Docker images and containers. Create the file in the root folder of your application and add the code from Listing 10-2 to it.

```
version: "3.0"
services:
  application:
    image:
      nextjs:latest
    ports:
      - "3000:3000"
    volumes:
      - .:/home/node/
    command:
      "npm run dev"
  jest:
    image:
      nextjs:latest
    volumes:
      - .:/home/node/
    command:
      "npx jest ./__tests__/mongoose/weather/services.test.ts --watchAll"
```

Listing 10-2: A basic *docker-compose.yml* file that defines the application and Jest services

Every *docker-compose.yml* file starts by setting the version of the Docker Compose specification used. Depending on the version, we can use different properties and values. We then define each service as a single property under services. As discussed, we want to have two services: our Next.js application running on port 3000 and the Jest service, which watches the *services.test.ts* file we created in Chapter 8 and reruns the tests as soon as we change a file. We limit the watch command to retest only the services. This limits the scope of the exercises, but of course, you can rerun all tests if you'd like.

Each service follows roughly the same structure. First we define the image from which Docker Compose should create each container. This can be an official distribution or a locally built one. We use the *nextjs* image in the latest version for both services. Then, instead of using the *--publishAll* flag, we map the ports directly from 3000 to 3000. By doing so, we can connect to the application's port 3000 from the host's port 3000.

With the *volumes* property, we synchronize the files and paths from the host system into the container. This is similar to the mapping we used in the *docker run* command, but instead of supplying an absolute path, we can use relative paths for the source. Here we map the whole local directory *./* into the container's working directory */home/node*. As before, we can edit the TypeScript files locally, and the application inside the container always uses the latest version of the files.

Until now, these properties have matched the command line arguments we used in the *docker run* command. Now we add the *command* property, which specifies the command that each container executes on startup. For the application service, we'll start Next.js with the usual *npm run dev* command, whereas the Jest service should call Jest directly through *npx*. Providing the path to the test file and the *--watchAll* flag causes Jest to rerun the tests when the source code changes.

Running the Containers

Start the multi-container app with the *docker compose up* command. The output should look similar to what is shown here:

```
$ docker compose up
[+] Running 2/2
 :: Container application-1    Created           0.0s
 :: Container jest-1    Recreated           0.4s
Attaching to application-1, jest-1
application-1 |
application-1 | > refactored-app@0.1.0 dev
application-1 | > next dev
application-1 |
application-1 | ready - started server on 0.0.0.0:3000, URL:
application-1 | http://localhost:3000
jest-1        | PASS __tests__/mongoose/weather/services.test.ts
jest-1        | the weather services
jest-1        | API storeDocument
jest-1        | ✓ returns true (9 ms)
jest-1        | ✓ passes the document to Model.create() (6 ms)
```

```

jest-1 | API findByZip
jest-1 |   ✓ returns true (1 ms)
jest-1 |   ✓ passes the zip code to Model.findOne() (1 ms)
jest-1 | API updateByZip
jest-1 |   ✓ returns true (1 ms)
jest-1 |   ✓ passes the zip code and the new data to
jest-1 |     Model.updateOne() (1 ms)
jest-1 | API deleteByZip
jest-1 |   ✓ returns true (1 ms)
jest-1 |   ✓ passes the zip code Model.deleteOne() (1 ms)
jest-1 |
jest-1 | Test Suites: 1 passed, 1 total
jest-1 | Tests:      8 passed, 8 total
jest-1 |           0 total
jest-1 | Time:       4.059 s
jest-1 | Ran all test suites matching
jest-1 |   /.\/_tests_\mongoose\weather\services.test.ts/i.

```

The Docker daemon spins up all services. As soon as the application is ready, we see the status message from the Express.js server and can connect to it on the exposed port 3000. At the same time, the Jest container runs the tests for the weather services and reports that all are successful.

Rerunning the Tests

Now that we've started the Docker environment, let's verify that the command to look for changes in the code and rerun tests is working as intended. To do so, we need to modify the source code to trigger Jest. Therefore, we open the *mongoose/weather/service.ts* file and modify the contents by adding a blank line and then saving the file. Jest should rerun the test inside the container, as you can see from the output in Listing 10-3.

```

jest-1 | Ran all test suites matching
jest-1 |   /.\/_tests_\mongoose\weather\services.test.ts/i.
jest-1 |
jest-1 | PASS __tests__/mongoose/weather/services.test.ts
jest-1 | the weather services
jest-1 |   API storeDocument
jest-1 |     ✓ returns true (9 ms)
jest-1 |     ✓ passes the document to Model.create() (6 ms)
jest-1 |   API findByZip
jest-1 |     ✓ returns true (1 ms)
jest-1 |     ✓ passes the zip code to Model.findOne() (1 ms)
jest-1 |   API updateByZip
jest-1 |     ✓ returns true (1 ms)
jest-1 |     ✓ passes the zip code and the new data to
jest-1 |       Model.updateOne() (1 ms)
jest-1 |   API deleteByZip
jest-1 |     ✓ returns true (1 ms)
jest-1 |     ✓ passes the zip code Model.deleteOne() (1 ms)
jest-1 |
jest-1 | Test Suites: 1 passed, 1 total
jest-1 | Tests:      8 passed, 8 total

```

```
jest-1          |    0 total
jest-1          | Time:      7.089 s
jest-1          | Ran all test suites matching
jest-1          | /\.\/__tests__\/mongoose\/weather\/services.test.ts/i
```

Listing 10-3: Rerunning the tests on files changed with `jest --watchAll`

All tests continue to pass. Connect to <http://localhost:3000> and verify that your browser can still render the application.

Interacting with Docker Compose

Docker Compose provides a complete interface for managing microservice applications. You can see a list of available commands by running `docker compose --help`. The following are the most essential.

We use `docker compose ls` to get a list of all locally running Docker applications defined in *docker-compose.yml* files. The command returns the name and status of the application:

```
$ docker compose ls
```

To shut down all running services defined in the *docker-compose.yml* file in the current directory, run `docker compose kill`, which sends a `SIGKILL` command to the primary process inside each container:

```
$ docker compose kill
```

To kill the services with a more graceful `SIGTERM` command, use the following:

```
$ docker compose down
```

Instead of forcing a shutdown, this command gracefully removes all processes, containers, networks, and volumes created by `docker compose up`.

Summary

Using the Docker containerization platform makes it easy to deploy applications and use a microservice architecture. This chapter covered the building blocks of the Docker ecosystem: the host, the Docker daemon, Dockerfiles, images, and containers. Using Docker Compose and Docker volumes, you split your application into single, self-contained services.

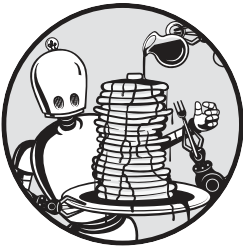
To unleash the full potential of Docker, read the official tutorials at <https://docs.docker.com/get-started/> or those at <https://docker-curriculum.com>. In the next chapter, you'll start to build the Food Finder application. This full-stack web application will build upon the knowledge you've gained in all previous chapters.

PART II

THE FULL-STACK APPLICATION

11

SETTING UP THE DOCKER ENVIRONMENT



In this part of the book, you'll build a full-stack application from scratch by using the knowledge you've acquired so far. While previous chapters explained parts of the technology stack, the remaining chapters focus on the code in more detail.

This chapter describes the application you'll build and walks you through configuring the environment using Docker. While I recommend reading previous chapters before you start writing code, the only real requirement is that you have Docker installed and running before moving on. Consult Chapter 10 for instructions on doing so.

NOTE

You can download the complete source code for the Food Finder application at <http://www.usemodernfullstack.dev/downloads/food-finder> and a ZIP file with only the required assets from <http://www.usemodernfullstack.dev/downloads/assets>.

The Food Finder Application

The Food Finder application shows a list of restaurants and their locations. The user can click these to see additional details about each location. In addition, they can log in to the app with their GitHub accounts by using OAuth so that they can maintain a wish list of locations.

Behind the scenes, we'll write this simple single-page application in TypeScript. After setting up the local environment, we'll build the backend and middleware with Next.js, Mongoose, and MongoDB, which we'll seed with initial data. Then we'll add GraphQL to expose an API layer through which we can access a user's wish list. To build the frontend, we'll use our knowledge of React components, Next.js pages, and routing. We'll also add an OAuth authorization flow with *next-auth* to let users log in with GitHub. Finally, we'll write automated tests with Jest to verify the integrity and stability of the application.

Building the Local Environment with Docker

Docker decouples the development environment from our local machine. We'll use it to create self-contained services for each part of our application. In the *docker-compose* file, we'll add one service for the backend, which provides the MongoDB database, and a second to run the Next.js application hosting the frontend and the middleware.

To start the development, create a new empty folder, *code*. This folder will serve as the application's root and contain all the code for the Food Finder application. Later in this chapter, we'll use the *create-next-app* helper command to add files to it.

Next, create an empty *docker-compose.yml* file and a *.docker* folder in this root folder. In the file, we will define the two services for our environment and store the seed data we need to create the container.

The Backend Container

The backend container provides nothing but the app's MongoDB instance. For this reason, we can use the official MongoDB image, which Docker can download automatically, from the Docker registry without creating a custom Dockerfile.

Seeding the Database

We want MongoDB to begin with a prefilled database that contains a valid set of initial datasets. This process is called seeding the database, and we can automate it by copying the seeding script *seed-mongodb.js* into the container's */docker-entrypoint-initdb.d/* directory on startup. The MongoDB image executes the scripts in this folder against the database defined in the *MONGO_INITDB_DATABASE* environment variable if there is no data in the container's */data/db* directory on startup.

Create a new folder, *foodfinder-backend*, in the *.docker* folder, and then copy into the newly created folder the *seed-mongodb.js* file from the *assets.zip* file you downloaded earlier. The seed file's content should look similar to Listing 11-1.

```
db.locations.insert([
  {
    address: "6220 Avenue U",
    zipcode: "NY 11234",
    borough: "Brooklyn",
    cuisine: "Cafe",
    grade: "A",
    name: "The Roasted Bean",
    on_wishlist: [],
    location_id: "56018",
  },
  --snip--
  {
    address: "405 Lexington Avenue",
    zipcode: "NY 10174",
    borough: "Manhattan",
    cuisine: "American",
    grade: "A",
    name: "The Diner At The Corner",
    on_wishlist: [],
    location_id: "63426",
  }
]);
```

Listing 11-1: The seed-mongodb.js file

You can see that the script interacts directly with a collection in the MongoDB instance that we'll set up in the next section. We use MongoDB's *insert* method to fill the database's *location* collection with the documents. Note that we are working with the *native* MongoDB driver to insert the documents instead of using Mongoose. We do so because Mongoose is not installed on the default MongoDB Docker image, and inserting the documents is a relatively simple task. Although we do not use Mongoose for seeding the database, the documents we insert need to match the schema we define with Mongoose later.

Creating the Backend Service

We can now define the backend service in the Docker setup. Add the code from Listing 11-2 into the empty *docker-compose.yml* file we created earlier.

```
version: "3.0"
services:
  backend:
    container_name: foodfinder-backend
    image: mongo:latest
    restart: always
```

```

environment:
  DB_NAME: foodfinder
  MONGO_INITDB_DATABASE: foodfinder
ports:
  - 27017:27017
volumes:
  - "./.docker/foodfinder-backend/seed-mongodb.js:
/docker-entrypoint-initdb.d/seed-mongodb.js"
  - mongodb_data_container:/data/db

volumes:
  mongodb_data_container:

```

Listing 11-2: The docker-compose.yml file with the backend service

We first define the container's name so that we can easily reference it later. As discussed earlier, we use the latest version of the official MongoDB image and specify that this container should always be restarted if it stops. Next, we use the environment variables to define the collections we'll use with MongoDB. We define two of those: `DB_NAME` points to the collection we'll use with Mongoose, and `MONGO_INITDB_DATABASE` points to the seed script. The scripts in `/docker-entrypoint-initdb.d/` use this latter collection by default.

We want the script to populate our application's database, so we set both variables to the same name, `foodfinder`, and thus we have a prefilled database for our Mongoose model.

Then we map and expose the container's internal port 27017 to the host's port 27017 so that the MongoDB instance is accessible to the application at `mongodb://backend:27017/foodfinder`. Notice that the connection string contains the service name, the port, and the database. Later, we store this connection string in the environment variables and use it to connect to the database from the middleware. Finally, we map and copy the seed script to the setup location and save the database data from `/data/db` into the Docker volume `mongodb_data_container`. Because we want to split the string across two lines, we need to wrap it in double quotes (") according to the YAML conventions.

Now complete the Docker setup with `docker compose up`:

```

$ docker compose up
[+] Running 2/2
  :: Network foodfinder_default          Created           0.1s
  :: Container foodfinder-backend        Created           0.3s
Attaching to foodfinder-backend

foodfinder-backend | /usr/local/bin/docker-entrypoint.sh: running /docker
                    /entrypoint-initdb.d/seed-mongodb.js

```

The output shows us that the Docker daemon successfully created the `foodfinder-backend` container and that the seeding script was executed during startup. Instead of going through the hassle of installing and maintaining MongoDB locally or finding a free or low-cost cloud instance, we've

added MongoDB to our project with just a few lines of code in the *docker-compose* file.

Stop the container with CTRL-C and remove it with `docker compose down`:

```
$ docker compose down
[+] Running 2/2
  :: Container foodfinder-backend          Removed          0.0s
  :: Network foodfinder_default            Removed
```

Now we can add the frontend container.

The Frontend Container

Now we'll create the containerized infrastructure for the frontend and middleware. Our approach will involve using `create-next-app` to scaffold the Next.js application, as we did in Chapter 5, relying on the official Node.js Docker image to decouple the application from any local Node.js installation.

As we'll execute all Node.js-related commands inside this container, we technically don't even need Node.js installed on our local machine; nor must we make sure the Node.js versions we use comply with Next.js's requirements. Also, npm might install packages that are optimized for the operating system on which it is running, so by using npm inside the container, we ensure that npm installs the correct versions for Linux.

Nonetheless, we'll want Docker to synchronize the Node.js *modules* folder to our local system. This will allow our IDE to automatically use the installed dependencies, such as the TypeScript compiler and ESLint. Let's start by creating a minimal Dockerfile.

Creating the Application Service

We add the combined frontend and middleware service to our Docker setup by placing the code from Listing 11-3 into the `services` property of the project's *docker-compose.yml* file.

```
--snip--
services:

  application:
    container_name: foodfinder-application
    image: node:lts-alpine
    ports:
      - "3000:3000"
    volumes:
      - ./code:/home/node/code
    working_dir: /home/node/code/
    depends_on:
      - backend
    environment:
      - HOST=0.0.0.0
```

```
        - CHOKIDAR_USEPOLLING=true
        - CHOKIDAR_INTERVAL=100
    tty: true
    backend:
--snip--
```

Listing 11-3: The docker-compose.yml file with the backend and application service

The service for the Food Finder application follows the same structure as the service for the backend. First we set the container's name. Then we define the image to be used for this particular service. While the backend service used the official MongoDB image, we now use the official Node.js image with the current LTS version running on Alpine Linux, a lightweight Linux distribution that requires significantly less memory than a Debian-based image.

We then expose and map port 3000, making the application available on `http://localhost:3000`, and map the local application's code directory into the container. Next, we set the working directory to the `code` directory. We specify that our container requires a running backend service, because the Next.js application will need a working connection to the MongoDB instance. In addition, we add environment variables. In particular, `chokidar` supports hot-reloading for the Next.js code. Finally, setting the `tty` property to `true` makes the container provide an interactive shell instead of shutting down. We'll need the shell to execute commands inside the container.

Installing Next.js

With both services in place, we can now install Next.js inside the container. To do so, we need to start the container with `docker compose up`:

```
$ docker compose up

[+] Running 3/3
  :: Network foodfinder_default          Created           0.1s
  :: Container foodfinder-backend        Created           0.3s
  :: Container foodfinder-application    Created           0.3s
Attaching to foodfinder-application, foodfinder-backend
--snip--
foodfinder-application | Welcome to Node.js ...
--snip--
```

Compare this command line output with the previous `docker compose up` output. You should see that the application container started successfully and that it runs a Node.js interactive shell.

Now we can use `docker exec` to execute commands inside the running container. Doing so has two main advantages. First, we don't need any particular version of Node.js (or any version at all) on our local machine. Second, we run the Node.js application and `npm` commands on the Node.js Linux Alpine image so that the dependencies will be optimized for Alpine instead of for our host system.

To run `npm` commands inside the container, use `docker exec -it foodfinder-application` followed by the command to run. The Docker

daemon connects to the terminal inside the container and executes the provided command in the application container's working directory, `/home/node/code`, which we set previously. Let's install the Next.js application there using the `npx` command discussed in Chapter 5:

```
/home/node/code# docker exec -it foodfinder-application \
npx create-next-app@latest foodfinder-application \
--typescript --use-npm
Need to install the following packages:
  create-next-app
Ok to proceed? (y)
✓ Would you like to use ESLint with this project? ... No / Yes
Creating a new Next.js app in /home/node/code/foodfinder-application.
```

Success! Created `foodfinder-application` at `/home/node/code/foodfinder-application`

We set the project name to *foodfinder-application* and accept the defaults. The rest of the output should look familiar to you.

As soon as the scaffolding is done, we can start the Next.js application with `npm run dev`. If you visit `http://localhost:3000` in your browser, you should see the familiar Next.js splash screen. The *foodfinder-application* folder should be mapped into the local *code* folder, so we can edit the Next.js-related files locally.

Adjusting the Application Service for Restarts

Currently, connecting to the application container requires running `docker exec` after each restart through `docker compose up` and then calling `npm run dev` manually. Let's make two minor adjustments in our application service to allow for a more convenient setup. Modify the file to match Listing 11-4.

```
--snip--
services:
--snip--
  application:
--snip--
    volumes:
      - ./code:/home/node/code
    working_dir: /home/node/code/foodfinder-application
--snip--
    command: "npm run dev"
--snip--
```

Listing 11-4: The `docker-compose.yml` file to start Next.js automatically

First, change the `working_dir` property. Because we're working with Next.js, we set it to the Next.js application's root folder, `/home/node/code/foodfinder-application`, which contains the `package.json` file. Then we add the `command` property with a value of `npm run dev`. With these two modifications, each `docker compose up` call should instantly start the Next.js application. Try

starting the containers with `docker compose up`; the console output should show that Next.js runs and that it's available at `http://localhost:3000`:

```
$ docker compose up
[+] Running 3/3
  :: Network foodfinder_default          Created    0.1s
  :: Container foodfinder-backend        Created    0.3s
  :: Container foodfinder-application    Created    0.3s
Attaching to foodfinder-application, foodfinder-backend
foodfinder-application |
foodfinder-application | > foodfinder-application@0.1.0 dev
foodfinder-application | > next dev
foodfinder-application |
foodfinder-application | ready - started server on 0.0.0.0:3000,
foodfinder-application | url: foodfinder-application | http://localhost:3000
foodfinder-application | info - Loaded env from /home/node/code/foodfinder-
foodfinder-application | application/.env.local
```

If you visit `http://localhost:3000` in your browser, you should see the Next.js splash screen without having to start the Next.js application manually.

Note that, if you're using Linux or macOS without being the administrator or root user, you'll need to adjust the application service and the startup command. Because the Docker daemon runs as a root user by default, all files it creates require root privileges. Your regular user doesn't have those and cannot access those files. To avoid these possible issues, modify your setup so that the Docker daemon transfers the ownership to your user. Start by adding the code in Listing 11-5 to the application service in the *docker-compose* file.

```
services:
--snip--
  application:
--snip--
    user: ${MY_USER}
--snip--
```

Listing 11-5: The docker-compose.yml file with the user property

We add the user property to the application service and use the environment variable `MY_USER` as the property's value. Then we modify the docker compose commands so that, on startup, we add the current user's user ID and group ID to this environment variable. Instead of a plain `docker compose up` call, we use the following code:

```
MY_USER=$(id -u):$(id -g) docker compose up
```

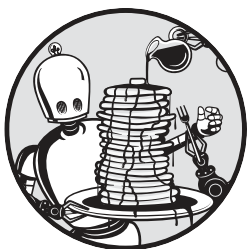
We use the `id` helper program to save the user ID and group ID in the format `userid:groupid` to our environment variable, which the *docker-compose* file then picks up. The `-u` flag returns the user ID, and the `-g` flag returns the group ID.

Summary

We've set up our local development environment with Docker containers. With the *docker-compose.yml* file we created in this chapter, we decoupled the application development from our local host system. Now we can switch our host systems and, at the same time, ensure that the Food Finder application always runs with the same Node.js version. In addition, we added a container running our MongoDB server, to which we'll connect in the next chapter when we implement our application's middleware.

12

BUILDING THE MIDDLEWARE



The middleware is the software glue connecting the frontend we'll create later to the existing MongoDB instance in the backend container. In this chapter, we'll set up Mongoose, connect it to our database, and then create a Mongoose model for the application. In the next chapter, we'll complete the middleware by writing a GraphQL API.

This middleware is part of Next.js; hence, we'll work with the application container. But because the Docker daemon ensures that the files in our local application directory are instantly available within the working directory inside the application container, we can use our local code editor or IDE to modify files on our local machine. There is no need to connect to the container shell, let alone interact with `docker compose`; you should see all changes instantly on <http://localhost:3000>.

Configuring Next.js to Use Absolute Imports

Before we write our first line of code in Next.js, let's make a minor adjustment to the Next.js configuration. We want the paths of any module imports to be *absolute*, meaning they start from the application's root folder rather than the location of the file that is importing them. The imports in Listing 12-1, which come from the `pages/api/graphql.ts` file we created in Chapter 6, are examples of relative imports.

```
import { resolvers } from "../../graphql/resolvers";
import { typeDefs } from "../../graphql/schema";
```

Listing 12-1: The import statements in pages/api/graphql.ts

You should see that they start from the file's location, then go up two levels to the root folder, and finally find the `graphql` folder containing the `resolvers` and `schema` TypeScript files.

The more complex our application becomes, the more levels of nesting we'll have, and the more inconvenient we'll find this manual traversing of the directories up to the root folder. This is why we want to use absolute imports that start directly from the root folder, as shown in Listing 12-2.

```
import { resolvers } from "graphql/resolvers";
import { typeDefs } from "graphql/schema";
```

Listing 12-2: The absolute import statements for pages/api/graphql.ts

Notice that we don't need to traverse up to the root level before importing files. To achieve this, open the `tsconfig.json` file that create-next-app created in the application's code root directory, `code/foodfinder-application`, on your local machine, and add a line that sets the `baseUrl` to the root folder (Listing 12-3).

```
{
  "compilerOptions": {
    "baseUrl": ".",
    --snip--
  }
}
```

Listing 12-3: Using absolute URLs

Restart the application's container, as well as the Next.js application, with `docker compose restart foodfinder-application` in a new command line tab.

Connecting Mongoose

Now it's time to start working on the middleware. We'll begin by adding Mongoose to the application. Connect to the application's container terminal:

```
$ docker exec -it foodfinder-application npm install mongoose
```

Here we use `npm install mongoose` to install the package. As long as the containers are running, we don't need to rebuild the frontend image immediately, as we've installed the packages directly into the running container.

Writing the Database Connection

To connect the Next.js application to the MongoDB instance, we'll define the environment variable `MONGO_URI` and assign it a connection string that matches the backend's exposed port and location. Create a new `.env.local` file in the application's root directory, next to the `tsconfig.json` file, and add this line to it:

```
MONGO_URI=mongodb://backend:27017/foodfinder
```

Now we can connect the application to the MongoDB instance that the Docker container exposes on port 27017. Create a folder, *middleware*, in the root folder `code/foodfinder-application`. Here we'll place all the middleware-related TypeScript files. Create a new file, *db-connect.ts*, in this folder and paste in the code from Listing 12-4.

```
import mongoose, { ConnectOptions } from "mongoose";

const MONGO_URI = process.env.MONGO_URI || "";

if (!MONGO_URI.length) {
  throw new Error(
    "Please define the MONGO_URI environment variable (.env.local)"
  );
}

let cached = global.mongoose;

if (!cached) {
  cached = global.mongoose = { conn: null, promise: null };
}

async function dbConnect(): Promise<any> {

  if (cached.conn) {
    return cached.conn;
  }

  if (!cached.promise) {

    const opts: ConnectOptions = {
      bufferCommands: false,
      maxIdleTimeMS: 10000,
      serverSelectionTimeoutMS: 10000,
      socketTimeoutMS: 20000,
    };

    cached.promise = mongoose.connect(MONGO_URI, opts).then(
      () => {
        cached.conn = mongoose.connection;
      }
    );
  }

  return cached.promise;
}
```

```

        cached.promise = mongoose
            .connect(MONGO_URI, opts)
            .then((mongoose) => mongoose)
            .catch((err) => {
                throw new Error(String(err));
            });
    }

    try {
        cached.conn = await cached.promise;
    } catch (err) {
        throw new Error(String(err));
    }

    return cached.conn;
}

export default dbConnect;

```

Listing 12-4: The TypeScript code to connect the application to the database in db-connect.ts

We import the *mongoose* package and the *ConnectOptions* type, both of which we need to connect to the database. We then load the connection string from the environment variables and verify that the string is not empty.

Next, we set up our connection cache. We use a global variable to maintain the connection across hot-reloads and ensure that multiple calls to our *dbConnect* function always return the same connection. Otherwise, there is the risk that our application will create new connections during each hot-reload or on each call of the function, both of which would fill up our memory quickly. If there's no cached connection, we initialize it with a dummy object.

We create the asynchronous function *dbConnect*, which actually opens and handles the connection. The database is remote and not instantly available, so we use an async function that we export as the module's default function. Inside the function's body, we first check for an already existing cached connection and directly return any existing ones. Otherwise, we create a new one. Therefore, we define the connection options, and then we create a new connection; here, we use the promise pattern to remind us of the two possible ways to handle asynchronous calls. Finally, we await the connection to be available, and then return the Mongoose instance.

To open a cached connection to MongoDB through Mongoose, we can now import the *dbConnect* function from the *middleware/db-connect* module and await the Mongoose connection.

Fixing the TypeScript Warning

In your IDE, you should immediately see that TSC warns us about using *global.mongoose*. A closer look at the message, *Element implicitly has an 'any' type because type 'typeof globalThis' has no index signature.ts (7017)*, tells us that we need to add the *mongoose* property to the *globalThis* object.

As we discussed in Chapter 3, we use the *custom.d.ts* file to define custom global types. Create a new file, *custom.d.ts*, next to the *middleware* folder in the root directory. As soon as you paste the code from Listing 12-5 into it, the global namespace should contain the *mongoose* property typed as *mongoose*, and TSC can find it.

```
import mongoose from "mongoose";

declare global {
  var mongoose: mongoose;
}
```

Listing 12-5: The code in custom.d.ts used to define the custom global type mongoose

With the custom global type definition in place, the TSC should no longer complain about the missing type definition for *global.mongoose*. We can move on to create the *Mongoose* model for our full-stack application.

The Mongoose Model

Our application has one database containing a collection of documents representing location data, as you saw in the seed script from Chapter 11. We'll create a *Mongoose* model for this location collection. In Chapter 7, you learned that this requires having an interface to type the documents for TypeScript, a schema to describe the documents for the model, a type definition, and a set of custom types to define the *Mongoose* model. In addition, we'll create a set of custom types to perform the CRUD operations on the locations model for the application.

Create a *mongoose* folder with the subfolder *locations* next to the *middleware* folder in Next.js's root directory. The *mongoose* folder will host all files relevant to *Mongoose* in general, and the *locations* folder will contain all files specific to the location model.

Creating the Schema

In Chapter 7, you learned that the schema describes the structure of a database's documents and that you need to create a TypeScript interface before creating a schema so that you can type the schema and model accordingly. Technically, in versions of *Mongoose* later than 6.3.1, we don't need to define this interface by ourselves. Instead, we can automatically infer the interface as a type from the schema. Create the file *schema.ts* inside the *mongoose/locations* folder and paste the code from Listing 12-6 into it.

```
import { Schema, InferSchemaType } from "mongoose";

export const LocationSchema: Schema = new Schema<LocationType>({
  address: {
    type: "String",
    required: true,
  },
},
```

```

    street: {
      type: "String",
      required: true,
    },
    zipcode: {
      type: "String",
      required: true,
    },
    borough: {
      type: "String",
      required: true,
    },
    cuisine: {
      type: "String",
      required: true,
    },
    grade: {
      type: "String",
      required: true,
    },
    name: {
      type: "String",
      required: true,
    },
    on_wishlist: {
      type: ["String"],
      required: true,
    },
    location_id: {
      type: "String",
      required: true,
    },
  },
});

```

```
export declare type LocationType = InferSchemaType<typeof LocationSchema>;
```

Listing 12-6: The mongoose/locations/schema.ts file

We import the Schema constructor and InferSchemaType, the function for inferring the schema type, both of which are part of the Mongoose module. Then we define and directly export the schema. The schema itself is straightforward. A document in the location collection has a few self-explanatory properties that are all typed as strings except for the `on_wishlist` property, which is an array of strings. To keep the application simple, we will store the IDs of users who added a particular location to their wish list directly in a location's document instead of creating a new Mongoose model and MongoDB document for each user's wish list. This isn't a great design for a real application, but it's fine for our purposes. Lastly, we infer and export the `LocationType` directly from the schema instead of creating the interface manually.

Creating the Location Model

With the schema and required interface in place, it's time to create the model. Create the file *model.ts* in the *mongoose/location* folder and paste the code from Listing 12-7 into it.

```
import mongoose, { model } from "mongoose";
import { LocationSchema, LocationType } from "mongoose/locations/schema";

export default mongoose.models.locations ||
  model<LocationType>("locations", LocationSchema);
```

Listing 12-7: The mongoose/locations/model.ts file

After importing the required dependencies from the Mongoose package, we import the *LocationSchema* and *LocationType* from the *schema.ts* file we created previously. Then we use these to create and export our locations model, unless there is already a model called *locations* initialized and present. In this case, we return the existing one.

At this point, we've successfully created the Mongoose model and connected it to the database. We can now access the MongoDB instance and create, read, update, and delete documents in the locations collection through Mongoose's API.

To test that everything is working, try creating a temporary REST API that initializes a connection to the database and then queries all documents through the model. You can make this new file, *test-middleware.ts*, in the application's *pages/api* folder and paste the code from Listing 12-8 into it.

```
import type { NextApiRequest, NextApiResponse } from "next";

import dbConnect from "middleware/db-connect";
import Locations from "mongoose/locations/model";

export default async function handler(
  req: NextApiRequest, res: NextApiResponse<any>
) {
  await dbConnect();
  const locations = await Locations.find({});
  res.status(200).json(locations);
}
```

Listing 12-8: A temporary REST API to test the database connection

This API imports required dependencies from Next.js, the *dbConnect* function, and the *locations* model we created earlier. In the asynchronous API handler, it calls the *dbConnect* function and waits until Mongoose connects to the database. Then it calls Mongoose's *find* API on the *locations* model with an empty filter object. Once it receives the locations, the API handler will send them to the client.

If you open *http://localhost:3000/api/test-middleware*, you should see a JSON object with all available locations, similar to Figure 12-1.

```
localhost:3000/test-middleware
[{"_id":"63dfcf731356eb0eadb5b","address":"4220 Avenue U","zipcode":"NY 11234","borough":"Brooklyn","cuisine":"Cafe","grade":"A","name":"The Roasted Bean","on_wishlist":[],"location_id":"56018"},{"_id":"63dfcf731356eb0eadb5f","address":"103-05 37 Avenue","zipcode":"NY 11368","borough":"Queens","cuisine":"Japanese","grade":"A","name":"From Tokyo With Love","on_wishlist":[],"location_id":"62432"},{"_id":"63dfcf731356eb0eadb5f","address":"5408 Avenue W","zipcode":"NY 11234","borough":"Brooklyn","cuisine":"Hamburgers","grade":"A","name":"Happy Bunny Burgers","on_wishlist":[],"location_id":"12340"},{"_id":"63dfcf731356eb0eadb5f","address":"8825 Astoria Boulevard","zipcode":"NY 11369","borough":"Queens","cuisine":"American","grade":"B","name":"Fries On The Boulevard","on_wishlist":[],"location_id":"56151"},{"_id":"63dfcf731356eb0eadb5f","address":"1 East 66 Street","zipcode":"NY 10065","borough":"Manhattan","cuisine":"Ice Cream","grade":"B","name":"Frozen & Sweet","on_wishlist":[],"location_id":"61322"},{"_id":"63dfcf731356eb0eadb60","address":"1346 E 84th St","zipcode":"NY 11234","borough":"Bronx","cuisine":"Bakery","grade":"A","name":"Happy Bread","on_wishlist":[],"location_id":"75445"},{"_id":"63dfcf731356eb0eadb61","address":"531 Myrtle Avenue","zipcode":"NY 11205","borough":"Brooklyn","cuisine":"Hamburgers","grade":"C","name":"White & Black","on_wishlist":[],"location_id":"62344"},{"_id":"63dfcf731356eb0eadb62","address":"2351 Ralph Ave","zipcode":"NY 11234","borough":"Brooklyn","cuisine":"Italian","grade":"A","name":"The Artisan's Oven","on_wishlist":[],"location_id":"56442"},{"_id":"63dfcf731356eb0eadb63","address":"79-23 63 Road","zipcode":"NY 11374","borough":"Queens","cuisine":"Swedish/Kosher","grade":"B","name":"The Kosher Place","on_wishlist":[],"location_id":"56048"},{"_id":"63dfcf731356eb0eadb64","address":"6405 11 Avenue","zipcode":"NY 11219","borough":"Brooklyn","cuisine":"American","grade":"B","name":"Bellyful Of Fries","on_wishlist":[],"location_id":"56649"},{"_id":"63dfcf731356eb0eadb65","address":"1839 Hudson Avenue","zipcode":"NY 11226","borough":"Brooklyn","cuisine":"Ice Cream","grade":"B","name":"Mario's Ice Cream","on_wishlist":[],"location_id":"56731"},{"_id":"63dfcf731356eb0eadb66","address":"2380 Southern Boulevard","zipcode":"NY 10460","borough":"Bronx","cuisine":"Vietnamese","grade":"B","name":"Saigon","on_wishlist":[],"location_id":"57217"},{"_id":"63dfcf731356eb0eadb67","address":"7715 18 Avenue","zipcode":"NY 11214","borough":"Brooklyn","cuisine":"Vietnamese","grade":"B","name":"The Hanoi Cafe","on_wishlist":[],"location_id":"57433"},{"_id":"63dfcf731356eb0eadb68","address":"709 Kings Highway","zipcode":"NY 11223","borough":"Brooklyn","cuisine":"Chinese","grade":"B","name":"The Hot Pot","on_wishlist":[],"location_id":"60045"},{"_id":"63dfcf731356eb0eadb69","address":"1269 Sutter Avenue","zipcode":"NY 11208","borough":"Brooklyn","cuisine":"Chinese","grade":"B","name":"Noodle Kitchen","on_wishlist":[],"location_id":"58429"},{"_id":"63dfcf731356eb0eadb6a","address":"245-15 Hillside Avenue","zipcode":"NY 11054","borough":"Queens","cuisine":"American","grade":"B","name":"Street Kitchen","on_wishlist":[],"location_id":"59400"},{"_id":"63dfcf731356eb0eadb6b","address":"351 West 57 Street","zipcode":"NY 10019","borough":"Manhattan","cuisine":"Irish","grade":"B","name":"Sean's Bar","on_wishlist":[],"location_id":"51861"},{"_id":"63dfcf731356eb0eadb6c","address":"203 Church Avenue","zipcode":"NY 11218","borough":"Brooklyn","cuisine":"Ice Cream","grade":"B","name":"Iceing","on_wishlist":[],"location_id":"60076"},{"_id":"63dfcf731356eb0eadb6d","address":"6909 3 Avenue","zipcode":"NY 11209","borough":"Brooklyn","cuisine":"Swedish","grade":"B","name":"Nordic","on_wishlist":[],"location_id":"61390"},{"_id":"63dfcf731356eb0eadb6e","address":"784 Prospect Park West","zipcode":"NY 11215","borough":"Brooklyn","cuisine":"American","grade":"C","name":"John's Kitchen","on_wishlist":[],"location_id":"61606"},{"_id":"63dfcf731356eb0eadb6f","address":"759 Broadway","zipcode":"NY 10003","borough":"Manhattan","cuisine":"Deli/Casual","grade":"C","name":"The Bell At The Corner","on_wishlist":[],"location_id":"61708"},{"_id":"63dfcf731356eb0eadb70","address":"119-08 20 Avenue","zipcode":"NY 11354","borough":"Queens","cuisine":"Deli/Casual","grade":"C","name":"Queens Deli","on_wishlist":[],"location_id":"61618"},{"_id":"63dfcf731356eb0eadb71","address":"3406 10 Street","zipcode":"NY 11106","borough":"Queens","cuisine":"Deli/Casual","grade":"C","name":"Bagels Plus","on_wishlist":[],"location_id":"61998"},{"_id":"63dfcf731356eb0eadb72","address":"512 East 74 Street","zipcode":"NY 10021","borough":"Manhattan","cuisine":"American","grade":"C","name":"Greasy Food","on_wishlist":[],"location_id":"61521"},{"_id":"63dfcf731356eb0eadb73","address":"730 Columbus Avenue","zipcode":"NY 10023","borough":"Manhattan","cuisine":"American","grade":"C","name":"Movie Central","on_wishlist":[],"location_id":"62274"},{"_id":"63dfcf731356eb0eadb74","address":"502 Amsterdam Avenue","zipcode":"NY 10024","borough":"Manhattan","cuisine":"Chicken","grade":"C","name":"Janet's Kitchen","on_wishlist":[],"location_id":"62098"},{"_id":"63dfcf731356eb0eadb75","address":"18 West Houston Street","zipcode":"NY 10012","borough":"Manhattan","cuisine":"American","grade":"C","name":"The Corner Grocery","on_wishlist":[],"location_id":"62244"},{"_id":"63dfcf731356eb0eadb76","address":"60 Wall Street","zipcode":"NY 10005","borough":"Manhattan","cuisine":"Turkish","grade":"C","name":"The Doner","on_wishlist":[],"location_id":"62715"},{"_id":"63dfcf731356eb0eadb77","address":"195 East 56 Street","zipcode":"NY 11203","borough":"Brooklyn","cuisine":"Greek","grade":"C","name":"Cafe At The Club","on_wishlist":[],"location_id":"62849"},{"_id":"63dfcf731356eb0eadb78","address":"7020 Avenue U","zipcode":"NY 11234","borough":"Brooklyn","cuisine":"Swedish/Kosher","grade":"A","name":"Gourmet Bagels","on_wishlist":[],"location_id":"56483"},{"_id":"63dfcf731356eb0eadb79","address":"56 Court Street","zipcode":"NY 11201","borough":"Brooklyn","cuisine":"Donuts","grade":"A","name":"Bally's Donuts","on_wishlist":[],"location_id":"63098"}]
```

Figure 12-1: The API to test the middleware returns a JSON object with all locations stored in the database.

You've successfully created the Mongoose model and run your first database query.

The Model's Services

Chapter 6 discussed how we usually abstract database CRUD operations into service calls to simplify the implementation of GraphQL APIs down the line. This is what we'll do now, and as a first step, let's outline the required functionality.

We need one public service that queries all available locations so that they can be displayed in the app's overview page. To display a location's details, we need another public service that can find a specific location. We'll opt to use the location's ID as a parameter for the service and then look up the location by ID. To handle the wish list functionality, we need a service that can update a user's wish list, as well as another service that we can use to decide whether a given location is currently on the user's wish list; depending on the result, we'll display either an Add To or Remove From button.

To design the service calls that find and return locations, we'll create one public function for each public API and a unified internal function, `findLocations`, that calls Mongoose's `find` function. The public APIs construct the filter object that Mongoose uses to filter the documents in the collection. In other words, it creates the database query. Also, it sets up additional options we'll pass to the Mongoose API. This design should reduce the amount of code we need to write and prevent repetition.

Creating the Location Service's Custom Types

You may have noticed that we'll need two custom types for the parameters to the unified `findLocations` function. One parameter defines the properties for a find operation related to the wish list, and one is a location's ID. Create the file `custom.d.ts` in the `mongoose/location` folder to define these types, as shown in Listing 12-9.

```
export declare type FilterLocationType = {
  location_id: string | string[];
};

export declare type FilterWishlistType = {
  on_wishlist: {
    $in: string[];
  };
};
```

Listing 12-9: The `mongoose/locations/custom.d.ts` file

We define and directly export these two custom types. `FilterLocationType` is straightforward. It defines an object with one property, the location's ID, which is either a string or an array of strings. We use it to find a location by its ID. The second type is `FilterWishlistType`, which we'll use to find all locations that contain the user's ID in their `on_wishlist` property. We set the value for Mongoose's `$in` operator as an array of strings.

Creating the Location Services

Now that we've created custom types for the services, we can implement them. As usual, we create a file `services.ts` in the `mongoose/location` folder and add the code from Listing 12-10 to it.

```
import Locations from "mongoose/locations/model";
import {
  FilterWishlistType,
  FilterLocationType,
} from "mongoose/locations/custom";
import { LocationType } from "mongoose/locations/schema";
import { QueryOptions } from "mongoose";

async function findLocations(
  filter: FilterLocationType | FilterWishlistType | {}
): Promise<LocationType[] | []> {
  try {
    let result: Array<LocationType | undefined> = await Locations.find(
      filter
    );
    return result as LocationType[];
  } catch (err) {
    console.log(err);
  }
}
```

```

    }
    return [];
  }

export async function findAllLocations(): Promise<LocationType[] | []> {
  let filter = {};
  return await findLocations(filter);
}

export async function findLocationsById(
  location_ids: string[]
): Promise<LocationType[] | []> {
  let filter = { location_id: location_ids };
  return await findLocations(filter);
}

export async function onUserWishlist(
  user_id: string
): Promise<LocationType[] | []> {
  let filter: FilterWishlistType = {
    on_wishlist: {
      $in: [user_id],
    },
  };
  return await findLocations(filter);
}

export async function updateWishlist(
  location_id: string,
  user_id: string,
  action: string
) : Promise<LocationType | null | {}>
{
  let filter = { location_id: location_id };
  let options: QueryOptions = { upsert: true, returnDocument: "after" };
  let update = {};

  switch (action) {
    case "add":
      update = { $push: { on_wishlist: user_id } };
      break;
    case "remove":
      update = { $pull: { on_wishlist: user_id } };
      break;
  }

  try {
    let result: LocationType | null = await Locations.findOneAndUpdate(
      filter,
      update,
      options
    );
    return result;
  }
}

```

```

    } catch (err) {
      console.log(err);
    }
    return {};
  }
}

```

Listing 12-10: The `mongoose/locations/services.ts` file

After importing dependencies, we create the function that will actually call Mongoose's `find` API on the model and await the data from the database. This function will query the database for all public services that use `find`, so it's the foundation of all our services. Its one parameter, the `filter` object, can be passed to the model's `find` function to retrieve the documents that match the filter. The filter is either an empty object that returns all locations or one of our custom types, `FilterLocationType` or `FilterWishlistType`. As soon as we have the data from the database, we cast it to the `LocationType` and then return it. If there is an error, we log it and then return an empty array to match the defined return types: either an array of `LocationTypes` or an empty array.

The following three functions are the public services, which will provide database access to other TypeScript modules and the user interface. All follow the same structure. First, within the `findLocationsById` function, we set the `filter` object to a particular parameter. Then we call the `findLocations` function with this service-specific filter object. Because every service calls the same function, services also have the same return signature, and each returns an array of locations or an empty array. The first uses an empty object. Hence, it filters for nothing and instead returns all documents from the collection. The function `findLocationsById` uses the `FilterLocationType` and returns the documents that match the given location IDs.

The next function, `onUserWishlist`, uses a slightly more complex filter object. It has the type `FilterWishlistType`, and we pass it to the `findLocations` function to get all locations whose `on_wishlist` array contains the given user ID. Note that we type the filter objects explicitly upon declaration. This deviates from the advice given in Chapter 3, but we do it here to ensure that TSC verifies the object properties, as it cannot infer the types from their usage in this case.

Finally, we implement the `updateWishlist` function. It is slightly different from the previous ones, but the overall structure should look familiar. Again, we build the filter object from the first parameter, and we use the second one, the user ID, to update the `on_wishlist` array. Unlike in previous functions, however, we use another parameter to specify whether we want to add or remove the user ID to or from the array. Using a `switch/case` statement here is a convenient way to reduce the number of exposed services. Depending on the `action` parameter, we fill the update object with either the `$push` operator, which adds the user ID to the `on_wishlist` array, or the `$pull` operator, which removes the user ID. We pass the object to Mongoose's `findOneAndUpdate` API to look for the first document that matches the filter, and we directly update the record and then return the updated document or an empty object.

Testing the Services

Let's use our temporary REST API to evaluate the services. Open the *test-middleware.ts* file we created earlier and update it with the code from Listing 12-11.

```
import type { NextApiRequest, NextApiResponse } from "next";
import dbConnect from "middleware/db-connect";

import { findAllLocations } from "mongoose/locations/services";

export default async function handler(
  req: NextApiRequest,
  res: NextApiResponse<any>
) {
  await dbConnect();
  const locations = await findAllLocations();
  res.status(200).json(locations);
}
```

Listing 12-11: The pages/api/test-middleware.ts file using the services

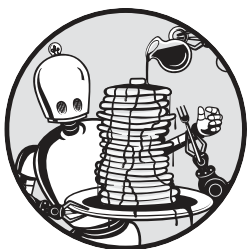
Instead of directly importing the model and using Mongoose's `find` method on it, we import the location services and query all locations with the `findAllLocations` service. If you open the API at <http://localhost:3000/api/test-middleware> in your browser, you should once again see a JSON object with all available locations.

Summary

We've successfully created the first part of the middleware. With the code in this chapter, we can use a Mongoose model to create, read, update, and delete documents in the MongoDB collection. To perform these actions, we set up the services we'll connect to our upcoming GraphQL API. In the next chapter, we'll delete the temporary testing API middleware and replace it with a proper GraphQL API.

13

BUILDING THE GRAPHQL API



In this chapter, you'll add a GraphQL API to the middleware by defining its schema, as well as resolvers for each query and mutation.

These resolvers will complement the Mongoose services created in Chapter 12. The queries will be public; however, we'll expose our mutations as protected APIs by adding an authorization layer via OAuth.

Unlike in the GraphQL API of Chapter 6, we'll follow a pattern of modularization to implement these schemas and resolvers. Instead of writing everything in one big file, we'll split the elements into separate files. Like using modules in modern JavaScript, this approach has the benefit of breaking down the code into smaller logical units, each with a clear focus. These units enhance the code's readability and maintainability.

Setting Up

We'll create the API's single-entry point `/api/graphql` with the Apollo server, which integrates into Next.js with the `@as-integrations/next` package. Start by installing the packages necessary for the GraphQL setup from the npm registry:

```
$ docker exec -it foodfinder-application npm install @apollo/server graphql graphql-tag @as-integrations/next \
```

After the installation is complete, create the folder `graphql/locations` next to the `middleware` folder in the application's root.

The Schemas

The first step to writing the schemas is to define the query and mutation typedefs, as well as any custom types we use for the schema. To do so, we'll split the schema into three files, `custom.gql.ts`, `queries.gql.ts`, and `mutations.gql.ts`, in the `graphql/locations` folder. Then we'll use an ordinary template literal to merge them into the final schema definition.

The Custom Types and Directives

Add the code from Listing 13-1 to the `custom.gql.ts` file to define the schema for the GraphQL queries.

```
export default `
  directive @cacheControl(maxAge: Int) on FIELD_DEFINITION | OBJECT
  type Location @cacheControl(maxAge: 86400) {
    address: String
    street: String
    zipcode: String
    borough: String
    cuisine: String
    grade: String
    name: String
    on_wishlist: [String] @cacheControl(maxAge: 60)
    location_id: String
  }
`;
```

Listing 13-1: The `graphql/locations/custom.gql.ts` file

The GraphQL API will return location objects from the Mongoose schema. Therefore, we must define a custom type representing these location objects. Create a custom Location type. To instruct the server to cache the retrieved values, set an `@cacheControl` directive for the whole custom type and a shorter one for the `on_wishlist` property because we expect this particular property to change frequently.

The Query Schema

Now add the code from Listing 13-2 to the *queries.gql.ts* file to define the schema for the queries.

```
export default `
  allLocations: [Location]!
  locationsById(location_ids: [String!]!): [Location]!
  onUserWishlist(user_id: String!): [Location]!
`;
```

Listing 13-2: The graphql/locations/queries.gql.ts file

We define a template literal with three GraphQL queries, all of which are entry points to the services we implemented for the Mongoose locations model in Chapter 12. The names and parameters are similar to those in the services, and the queries follow the GraphQL syntax you learned about in Chapter 6.

The Mutation Schema

To define the mutation schema, paste the code from Listing 13-3 into the *mutations.gql.ts* file.

```
export default `
  addWishlist(location_id: String!, user_id: String!): Location!
  removeWishlist(location_id: String!, user_id: String!): Location!
`;
```

Listing 13-3: The graphql/locations/mutations.gql.ts file

We create two mutations as template literals using GraphQL syntax: one for adding an item to the user's wish list and one for removing it. Both will use the *updateWishlist* function we implemented on the location services, so they require the *location_id* and the *user_id* as parameters.

Merging the Typedefs into the Final Schema

We've split the location schema into two files, one for the queries and one for the mutations, and placed their custom types in a third file; however, to initiate the Apollo server, we'll need a unified schema. Luckily, the typedefs are nothing more than template literals, and if we use template literal placeholders, the parser can interpolate these into a complete string. To accomplish this, create a new file, *schema.ts*, in the *graphql* folder and add the code from Listing 13-4.

```
import gql from "graphql-tag";

import locationTypeDefsCustom from "graphql/locations/custom.gql";
import locationTypeDefsQueries from "graphql/locations/queries.gql";
import locationTypeDefsMutations from "graphql/locations/mutations.gql";
```

```
export const typeDefs = gql`

  ${locationTypeDefsCustom}

  type Query {
    ${locationTypeDefsQueries}
  }

  type Mutation {
    ${locationTypeDefsMutations}
  }

`;
```

Listing 13-4: The graphql/schema.ts file

We import the `gql` tag from the *graphql-tag* package. Even though doing so is optional when working with the Apollo server, we keep the `gql` tag in front of our tagged template to ensure compatibility with all other GraphQL implementations. This also produces proper syntax highlighting in the IDE, which statically analyzes type definitions as GraphQL tags.

Next, we import the dependencies and schema fragments we'll use to implement the unified schema. Finally, we create a tagged template literal with the `gql` function, using template literal placeholders to merge the schema fragments into the schema skeleton. We add the custom Location type and then merge the queries' typedefs into the Query object and the Mutations into the mutation object and export the schema const as typedefs.

The GraphQL Resolvers

Now that we have the schema, we'll turn to the resolvers. We'll use a similar development pattern, writing the queries and mutations in separate files, then merging them into the single file we need for the Apollo server. Start by creating the *queries.ts* and *mutations.ts* files in the *graphql/locations* folder and then add the code from Listing 13-5 to *queries.ts*.

```
import {
  findAllLocations,
  findLocationsById,
  onUserWishlist,
} from "mongoose/locations/services";

export const locationQueries = {
  allLocations: async (_, any) => {
    return await findAllLocations();
  },
  locationsById: async (_, any, param: { location_ids: string[] }) => {
    return await findLocationsById(param.location_ids);
  },
};
```

```

    onUserWishlist: async (_, param: { user_id: string }) => {
      return await onUserWishlist(param.user_id);
    },
  },
};

```

Listing 13-5: The graphql/locations/queries.ts file

We import our services from the Mongoose folder and then create and export the location query object. The structure of each query follows the structure discussed in Chapter 6. We make one query for each service, and their parameters match those in the services.

For our mutations, add the code from Listing 13-6 to the *mutations.ts* file.

```

import { updateWishlist } from "mongoose/locations/services";

interface UpdateWishlistInterface {
  user_id: string;
  location_id: string;
}

export const locationMutations = {
  removeWishlist: async (
    _: any,
    param: UpdateWishlistInterface,
    context: {}
  ) => {
    return await updateWishlist(param.location_id, param.user_id,
      "remove"
    );
  },
  addWishlist: async (_, param: UpdateWishlistInterface, context: {}) => {
    return await updateWishlist(param.location_id, param.user_id, "add");
  },
};

```

Listing 13-6: The graphql/locations/mutations.ts file

Here we import only the `updateWishlist` function from our services. This is because we defined it as the single entry point for updating our documents, and we opted to use the third parameter, with the value `add` or `remove`, to distinguish between the two actions the mutation should perform. We also create the `UpdateWishlistInterface`, which we don't export. Instead, we'll use it inside this file to avoid repeating code when we define the interface for the functions' `param` argument.

As mutations, we create two functions at the `locationMutations` object, one for adding an item from a user's wish list and one for removing it. Both use the `updateWishlist` service and supply the value parameter corresponding to the action the user would like to take. The two mutations, `removeWishlist` and `addWishlist`, also take a third object called `context`. For now, it's an empty object, but in Chapter 15, we'll replace it with the

session information necessary to verify the identity of the user performing the action.

Create the final resolvers file, *resolvers.ts*, in the *graphql* folder and add the code from Listing 13-7 to it. This code will merge the mutation and query definitions.

```
import { locationQueries } from "graphql/locations/queries";
import { locationMutations } from "graphql/locations/mutations";

export const resolvers = {
  Query: {
    ...locationQueries,
  },
  Mutation: {
    ...locationMutations,
  },
};
```

Listing 13-7: The graphql/resolvers.ts file

In addition to the schema, we must pass the Apollo server an object containing all resolvers, as discussed in Chapter 6. To be able to do so, we must import the queries and mutations. Then we use the spread operator to merge the imported objects into the resolvers object, which we export. Now, with the schema and resolvers object available, we can create the API endpoint and instantiate the Apollo server.

Adding the API Endpoint to Next.js

When we discussed the differences between REST and GraphQL APIs, we pointed out that unlike REST, where every API has its own endpoint, GraphQL provides only one endpoint, typically exposed as */graphql*. To create this endpoint, we'll use the Apollo server's Next.js integration, as we did in Chapter 6.

Create the *graphql.ts* file in the *pages/api* folder and copy the code in Listing 13-8, which defines the API handler and its single entry point.

```
import { ApolloServer, BaseContext } from "@apollo/server";
import { startServerAndCreateNextHandler } from "@as-integrations/next";

import { resolvers } from "graphql/resolvers";
import { typeDefs } from "graphql/schema";
import dbConnect from "middleware/db-connect";

import { NextApiHandler, NextApiRequest, NextApiResponse } from "next";

❶ const server = new ApolloServer<BaseContext>({
  resolvers,
  typeDefs,
});
```

```

❷ const handler = startServerAndCreateNextHandler(server, {
  context: async () => {
    const token = {};
    return { token };
  },
});

❸ const allowCors =
  (fn: NextApiHandler) =>
  async (req: NextApiRequest, res: NextApiResponse) => {
    res.setHeader("Allow", "POST");
    res.setHeader("Access-Control-Allow-Origin", "*");
    res.setHeader("Access-Control-Allow-Methods", "POST");
    res.setHeader("Access-Control-Allow-Headers", "*");
    res.setHeader("Access-Control-Allow-Credentials", "true");

    if (req.method === "OPTIONS") {
      res.status(200).end();
    }
    return await fn(req, res);
  };

❹ const connectDB =
  (fn: NextApiHandler) =>
  async (req: NextApiRequest, res: NextApiResponse) => {
    await dbConnect();
    return await fn(req, res);
  };

export default connectDB(allowCors(handler));

```

Listing 13-8: The pages/api/graphql.ts file

We import all the elements we need to create the API handler: the Apollo server, a helper for the Apollo–Next.js integration, our resolvers, the GraphQL schema files, the function used to connect to the database, and the Next.js API helpers.

We create a new Apollo server with the resolvers and schema ❶. Then we use the Next.js integration helper ❷ to start the Apollo server and return a Next.js handler. The integration helper uses a serverless Apollo setup to smoothly integrate into the Next.js custom server instead of creating its own. In addition, we pass the context with an empty token to the handler. This is how we'll access the JWT we receive in the OAuth flow and pass it to the resolvers later.

Next, we create the wrapper functions discussed in Chapter 6 to add the CORS headers ❸ and ensure that we have a database connection on each API call ❹. We can safely do so because we set up our database connection in a way that returns the existing cached connection. Finally, we export the returned asynchronous wrapped handler.

Visit the Apollo sandbox at <http://localhost:3000/api/graphql> and run a few queries to test the GraphQL API before moving on to the next chapter.

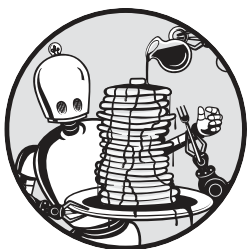
If you see the weather queries and mutations instead of the Food Finder's, clear your browser's cache and do a hard reload.

Summary

We've successfully added the GraphQL API to the middleware. With the code in this chapter, we can now use the Apollo sandbox to read and update values in the database. We've also already prepared the Apollo handler for authentication by providing it with an empty token. Now we're ready to use the JWT token we'll receive from the OAuth flow in Chapter 15 to protect the API's mutations. But before we add this authentication, let's build the frontend.

14

BUILDING THE FRONTEND



In this chapter, you'll build the frontend using React components and Next.js pages, discussed in Chapters 4 and 5. By the end, you'll have an initial version of the app to which you can add OAuth authentication.

Overview of the User Interface

Our application will consist of three Next.js pages. The *start page* will show the list of locations retrieved from the database. Each item in the list will link to its respective *location detail page*, whose URL we'll construct using the location's ID, like this: `/location/:location_id`. The third page is the user's *wish list page*. It resembles the start page and follows the same dynamic URL pattern as the location detail page, except it supplies the user's ID instead of the location's. This page shows only the locations already added to the wish list.

We must also consider what rendering strategy to use for each page. Because the content of the start page never changes, we'll use static site generation (SSG) to render the HTML on build time. Because the detail

page and wish list page will change based on the user's actions, we'll use static site rendering (SSR) to regenerate them upon every request.

Lastly, all three pages should have headers containing the logo and a link to the start page. When we add the OAuth data in the next chapter, we'll show the user's name, a link to the user's wish list, and the sign-in/sign-out button in the header as well.

To achieve this, we need to create the following React components:

- The locations list component, which will use the locations list item component to render the list of locations on the start page. Later, we'll use these same components to implement the list of locations on a user's wish list page.
- The overall layout component, header component, and logo component, which define the global layout of each page.
- The authentication element component, which lets users log in or out in the header.
- A universal button component we'll use for different tasks.

Let's begin with the components necessary for the start page.

The Start Page

We'll begin by crafting the smallest parts of the user interface and then use these to build the more complex components and pages. On the start page, we need the layout component, the locations list component, and the locations list item component, which is the smallest building block, so we'll start there.

Create the *components* folder in the application's root directory, next to the *middleware* folder. This is where we'll place all our React components, in their own folders.

The List Item

The locations list item component represents a single item in a list of locations. Create the *locations-list-item* folder and add two files, *index.tsx* and *index.module.css*, following the pattern we discussed in Chapter 5. Then add the code in Listing 14-1 to *index.module.css*. We'll use this CSS to style the component.

```
.root {
  background-color: #fff;
  border-radius: 5px;
  color: #1d1f21;
  cursor: pointer;
  list-style: none;
  margin: 0.5rem 0;
  padding: 0.5rem;
  transition: background-color 0.25s ease-in, color 0.25s ease-in;
  will-change: background-color, color;
}
```

```

.root:hover {
  background-color: rgba(0, 118, 255, 0.9);
  color: #fff;
}

.root h2 {
  margin: 0;
  padding: 0;
}

.root small {
  font-weight: 300;
  padding: 0 1rem;
}

```

Listing 14-1: The components/locations-list-item/index.module.css file

The CSS module uses dark letters on a white background. In addition, it adds a simple hover effect, causing the background to turn blue and the font color white when a user hovers over it. We remove the list marker and set the margin and padding accordingly.

Now add the code from Listing 14-2 to the *index.tsx* file.

```

import Link from "next/link";
import styles from "../index.module.css";
import { LocationType } from "mongoose/locations/schema";

interface PropsInterface {
  location: LocationType;
}

const LocationsListItem = (props: PropsInterface): JSX.Element => {
  const location = props.location;
  return (
    <>
      {location && (
        <li className={styles.root}>
          <Link href={` /location/${location.location_id}`}>
            <h2>
              {location.name}
              <small className={styles.details}>
                {location.cuisine} in {location.borough}
              </small>
            </h2>
          </Link>
        </li>
      )}
    </>
  );
};

export default LocationsListItem;

```

Listing 14-2: The components/locations-list-item/index.tsx file

You should be familiar with this file's structure from Chapter 5. First we import the *next/link* component, which we need to create a link to the detail page, the styles we just added, and the *LocationType* from the Mongoose schema.

We then define the *PropsInterface*, a private interface used for the component's properties object. The component has the usual *props* parameter whose structure defines the *PropsInterface* and returns a *JSX* element. These *props* hold the data in the *location* property, which we pass to the component through its *location* attribute. Finally, we define the *LocationsListItem* component and store it in a constant that we export at the end of the file.

In the component itself, we have a list item that contains a *Next.js* *Link* element linking to the location's detail page. These links use a dynamic URL pattern that incorporates the respective location's ID, so we create the link target to match */location/:location_id*. In addition, we render the location's name, cuisine, and borough values to the component. Keep in mind that until we create the page for the route */location/:location_id*, clicking those links will result in a *404* error page.

The Locations List

Using the list item component, we'll build the locations list. This component will loop through an array of locations and display them on the start page and wish list page. Create the *components/locations-list* folder and then add the files *index.tsx* and *index.module.css* to them. Copy the code in Listing 14-3 to the *index.module.css* file.

```
.root {
  margin: 0;
  padding: 0;
}
```

Listing 14-3: The components/locations-list/index.module.css file

The styles for the locations list component are simple; we remove the margin and padding from the component's root element. We create the component itself in Listing 14-4, which you should copy to *index.tsx*.

```
import LocationsListItem from "components/locations-list-item";
import styles from "../index.module.css";
import { LocationType } from "mongoose/locations/schema";

interface PropsInterface {
  locations: LocationType[];
}

const LocationsList = (props: PropsInterface): JSX.Element => {
  return (
    <ul className={styles.root}>
      { props.locations.map((location) => {
        return (
          <LocationsListItem
            location={location}

```

```

                                key={location.location_id}
                                />
                            );
                        }}}
                    </ul>
                );
    };

export default LocationsList;

```

Listing 14-4: The components/locations-list/index.tsx file

We import the `LocationsListItem` we just implemented, along with the module's styles and the `LocationType` from Mongoose's schema. We then define the component's `PropsInterface` to describe the component's props object. In the `LocationsList` component, we use the array `map` function to iterate over the location objects, rendering a `LocationsListItem` component for each array item and using the `location` attribute to pass the location details to the components. React requires that each item rendered in a loop have a unique ID. We use the location IDs for this purpose.

We can now create the start page and pass all available locations to this component. Later, we'll use the same component for the wish list page to return the locations on the user's wish list.

The Page

At this point, we have the components we need for the start page, which is a basic Next.js page. Save this page's global styles in *styles/globals.css* and its code in *pages/index.tsx*. Listing 14-5 contains the styles. Delete all other files from the *styles* directory. Those are default styles we don't need for the application.

```

html,
body {
  font-family: -apple-system, Segoe UI, Roboto, sans-serif;
  margin: 0;
  padding: 0;
}

* {
  box-sizing: border-box;
}

h1 {
  font-size: 3rem;
}

a {
  color: inherit;
  text-decoration: none;
}

```

Listing 14-5: The styles/globals.css file

We set a few global styles, such as the default font family, and change the box model to the more intuitive border-box for all elements. By using a border-box instead of a content-box, an element adopts whatever width we assign to it with the width property. Otherwise, the width property would define only the width of the content, and we'd need to add the border and padding to calculate the actual dimensions of the element on the page. We set the font families to the defaults for each operating system to ensure readability.

Now replace the existing content of the *pages/index.tsx* file with the code in Listing 14-6.

```
import Head from "next/head";
import type { GetStaticProps, InferGetStaticPropsType, NextPage } from "next";

import LocationsList from "components/locations-list";
import dbConnect from "middleware/db-connect";
import { findAllLocations } from "mongoose/locations/services";
import { LocationType } from "mongoose/locations/schema";

❶ const Home: NextPage = (
  props: InferGetStaticPropsType<typeof getStaticProps>
) => {

  ❷ const locations: LocationType[] = JSON.parse(props.data?.locations);
  let title = `The Food Finder - Home`;

  return (
    <div>
      <Head>
        <title>{title}</title>
        <meta name="description" content="The Food Finder - Home" />
      </Head>

      <h1>Welcome to the Food Finder!</h1>
      <LocationsList locations={locations} />
    </div>
  );
};

❸ export const getStaticProps: GetStaticProps = async () => {
  let locations: LocationType[] | [];
  try {
    await dbConnect();
    ❹ locations = await findAllLocations();
  } catch (err: any) {
    return { notFound: true };
  }
  ❺ return {
    props: {
      data: { locations: JSON.stringify(locations) },
    },
  };
};
```

```

    };
  };

  export default Home;

```

Listing 14-6: The pages/index.tsx file

We implemented the Next.js page, similar to the structure discussed in Chapter 5. First we import all dependencies; then we create the `NextPage` and store it in a constant that we export at the end of the file ❶.

The Next.js page's props object, the page properties, contains the data we return from the `getStaticProps` function ❷, discussed in Chapter 5. In this asynchronous function, we connect to the database ❸. As soon as the connection is ready, we call the service method to retrieve all locations ❹ and then pass them as a JSON string to the `NextPage` in the `data.locations` property of the props object. Next.js calls the `getStaticProps` function on build time and generates the HTML for this page only once. We can use this rendering method because the list of available locations never changes; it is static.

Then we retrieve the locations from the page properties ❺, parse the JSON string back to an array, and store the page title in a variable. We type the locations constant explicitly because TSC cannot easily infer the type. Then we construct the JSX. In the first step, we use the `next/head` component to set the page-specific metadata. Then we call the `locationList` component we previously implemented with the locations array in the locations attribute. By doing so, the `locationList` component renders all locations as an overview list.

As soon as you save the file, you should see, in the Docker command line, that Next.js recompiles the application. Open the web application at `http://localhost:3000` in your browser to see a list of locations similar to Figure 14-1.

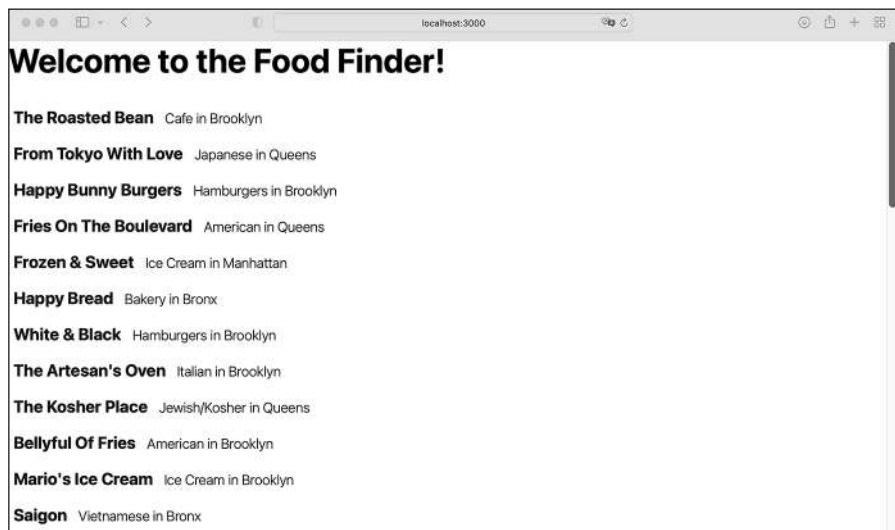


Figure 14-1: The start page showing all available locations

Now we'll move on to styling the frontend and adding basic global components, such as the application's header with the Food Finder logo.

The Global Layout Components

Now it's time to create the three global components. These include the overall layout component, which we'll use to format the start and wish list page content, a *sticky* header (which is always visible, "sticking" to the browser's upper edge), and the Food Finder logo to go in the header. Again, we'll start with the smallest units and then use those as building blocks for the overall components.

The Logo

The smallest component, the logo, is nothing more than a `next/image` component wrapped in a `next/link` element; when users click the logo image, they'll be redirected to the start page. Add a *header* folder to the *components* folder, then add a *logo* folder to the *header* folder and create two files there, *index.tsx* and *index.module.css*, into which you should paste the code in Listing 14-7.

```
.root {
  display: inline-block;
  height: 35px;
  position: relative;
  width: 119px;
}

@media (min-width: 600px) {
  .root {
    height: 50px;
    width: 169px;
  }
}
```

Listing 14-7: The `components/header/logo/index.module.css` file

These basic styles for the component's root element set the image's dimensions. We use a *mobile-first design pattern* by initially defining the styles to use on smaller screens and then, using a standard CSS media query, modifying them for screens bigger than 600px. We'll use a bigger image on bigger screens.

Now let's create the logo component. Create an *assets* subfolder in the Next.js *public* folder and place the *logo.svg* file extracted from *assets.zip* into it. Then add the code in Listing 14-8 to the logo's *index.tsx* file.

```
import Image from "next/image";
import Link from "next/link";
import logo from "/public/assets/logo.svg";
import styles from "./index.module.css";
```

```

const Logo = (): JSX.Element => {
  return (
    <Link href="/" passHref className={styles.root}>
      <Image
        src={logo}
        alt="Logo: Food Finder"
        sizes="100vw"
        fill
        priority
      />
    </Link>
  );
};

export default Logo;

```

Listing 14-8: The components/header/logo/index.tsx file

As usual, we import the dependencies and then create an exported constant that contains the JSX code. We don't pass any data to it through attributes or child elements; hence, we don't need to define the component's props object here.

We use a basic `next/image` inside a `next/link` element to link back to the start page and set the `next/image`'s attributes to fill the available space defined in the CSS file.

The Header

The header component will wrap the logo component we just created. Create the *index.tsx* file and *index.module.css* file in the *header* folder, then add the code in Listing 14-9 to the CSS file.

```

.root {
  background: white;
  border-bottom: 1px solid #eaeaea;
  padding: 1rem 0;
  position: sticky;
  top: 0;
  width: 100%;
  z-index: 1;
}

```

Listing 14-9: The components/header/index.module.css file

We use the CSS definitions `position: sticky` and `top: 0` to stick the header to the upper edge of the browser. Now the header will automatically stay there even when users scroll down the page; the page's content should scroll below the header because we set the header's `z-index`, placing the header in front of the other elements. You can think of the `z-index` as determining which floor of a building an element is on.

Listing 14-10 shows the code for the header component. Copy it into the component's *index.tsx* file.

```
import styles from "./index.module.css";
import Logo from "components/header/logo";

const Header = (): JSX.Element => {
  return (
    <header className={styles.root}>
      <div className="layout-grid">
        <Logo />
      </div>
    </header>
  );
};

export default Header;
```

Listing 14-10: The components/header/index.tsx file

We define a basic component that displays the logo. Then we wrap the imported Logo component in an element with a global layout-grid class, which we'll define in the next section.

The Layout

Currently, we have one Next.js page (the start page) and a header component. The easiest way to add the header to the page would be to import it into the Next.js page and place it directly into the JSX. However, we'll add two more pages to the app, the wish list page and the location detail page, so we want to avoid importing the header three times.

To streamline the overall app design, Next.js provides the concept of a *layout*, which is really just another component, and we can use it to add the header component as a sibling element to a page's content. Let's create a new layout component. First, to create this component's CSS file, add *layout.css* to the *styles* folder and paste the code in Listing 14-11 into it.

```
.layout-grid {
  align-items: center;
  display: flex;
  flex-direction: column;
  justify-content: space-between;
  margin: 0 auto;
  max-width: 800px;
  padding: 0 1rem;
  width: 100%;
}

@media (min-width: 600px) {
  .layout-grid {
    flex-direction: row;
    padding: 0 2rem;
  }
}
```

Listing 14-11: The styles/layout.css file

We use the mobile-first pattern once again to define a basic grid wrapper, setting the global padding and maximum width for the content area. We set the wrapper's left and right margins to auto, which centers the container, because the margins take up all available space between the fixed-width wrapper and the window's edges.

We use flexbox to set the direction of the wrapper's direct child elements to column, displaying them one on top of the next. Because the logo and all other upcoming header elements are direct children of an element with the layout-grid class, they are affected by the flexbox layout. In contrast, the location items aren't direct siblings. Hence, they won't change their direction when switching between screen sizes.

Then we use a media query to adjust the styles for screens whose width is greater than 600px. Here we increase the padding and change the layout order of the direct child elements. Instead of using column, we set it to row, and immediately we display the elements next to one another.

Because this is a global styles file and not a CSS module, Next.js won't automatically scope the class names. Hence, we prefix them with layout- and don't import the styles into the component before using them.

Now create a *layout* folder inside the *components* folder and add the *index.tsx* file to it with the component code in Listing 14-12.

```
import Header from "components/header";

interface PropsInterface {
  children: React.ReactNode;
}

const Layout = (props: PropsInterface): JSX.Element => {
  return (
    <>
      <Header />
      <main className="layout-grid">
        {props.children}
      </main>
    </>
  );
};
export default Layout;
```

Listing 14-12: The components/layout/index.tsx file

In the layout component, we define a private interface and the component with the usual structure. Inside the component, we add the Header and the main element that uses the global layout styles and acts as a wrapper for the children elements we'll pass to this component in the *_app.tsx* file.

Open the *_app.tsx* file and modify it as shown in Listing 14-13.

```
import "../styles/globals.css";
import "../styles/layout.css";
import type { AppProps } from "next/app";
import Layout from "components/layout";
```

```
export default function App({ Component, pageProps }: AppProps) {
  return (
    <Layout>
      <Component {...pageProps} />
    </Layout>
  );
}
```

Listing 14-13: The `pages/_app.tsx` file

First we add *layout.css* as a global style. As for the layout, we have only one layout component we'll use for all pages, and we import it here. Then we wrap our application, the pages, with the layout and pass the current page in the component's children property.

Now all our Next.js pages will follow the same structure: they'll have the Header component next to the main element containing the page's content. One advantage of following this pattern is that the component's state will be preserved across page changes and React component re-rendering.

Once Next.js has recompiled the application, try reloading the application at *http://localhost:3000* in your browser. It should look like Figure 14-2.

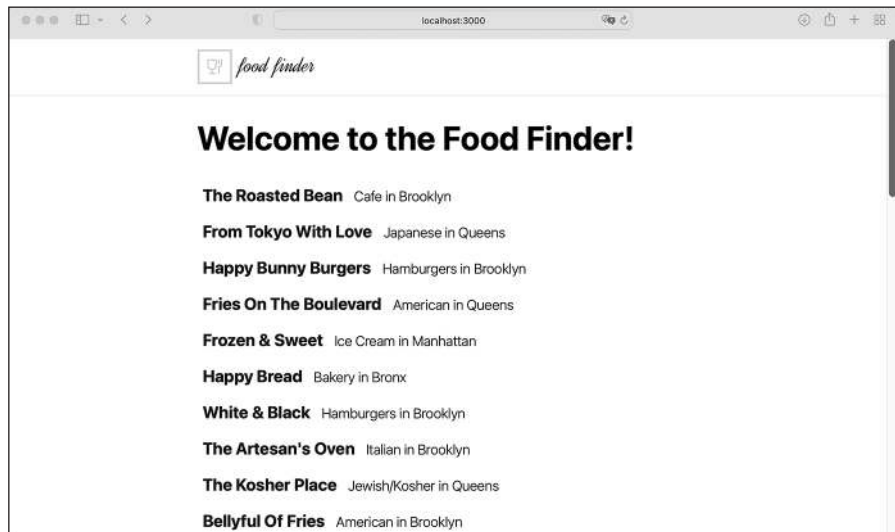


Figure 14-2: The start page with the header and layout component

You should now see the header, and the new layout component centers the content.

The Location Details Page

Our application now has a start page with a header and a list of all available locations. The list items link to their particular location's detail page because we added a `next/link` component to them, but those pages don't exist yet. If you click one of the links, you'll get a *404* error. To display the location details pages, we first need to implement the component that lists a particular location's details and then create a new Next.js page.

The Component

Let's start with the details component. Create the *location-details* folder in the *components* directory and add the *index.module.css* and *index.tsx* files to it. Then add the code from Listing 14-14 to the CSS module.

```
.root {
  margin: 0 0 2rem 0;
  padding: 0;
}
.root li {
  list-style: none;
  margin: 0 0 0.5rem 0;
}
```

Listing 14-14: The components/locations-details/index.module.css file

The styles for the component are basic. We remove the default margin and padding, as well as the list styles, and then add a custom margin at the end of each list item and root element.

To implement the location details component, add the code from Listing 14-15 to the *index.tsx* file in the *components/locations-details* folder.

```
import { LocationType } from "mongoose/locations/schema";
import styles from "./index.module.css";

interface PropsInterface {
  location: LocationType;
}

const LocationDetail = (props: PropsInterface): JSX.Element => {
  let location = props.location;
  return (
    <div>
      {location && (
        <ul className={styles.root}>
          <li>
            <b>Address: </b>
            {location.address}
          </li>
          <li>
            <b>Zipcode: </b>
            {location.zipcode}
          </li>
        </ul>
      )}
    </div>
  )
}
```

```

        <li>
          <b>Borough: </b>
            {location.borough}
        </li>
        <li>
          <b>Cuisine: </b>
            {location.cuisine}
        </li>
        <li>
          <b>Grade: </b>
            {location.grade}
        </li>
      </ul>
    )}
  </div>
);
};
export default LocationDetail;

```

Listing 14-15: The components/locations-details/index.tsx file

The locations detail component is structurally similar to the locations list item. Both take an object containing the location's data and add a specific set of properties to the returned JSX element. The main difference is in the JSX structure we create. Otherwise, we follow the known pattern, importing the required styles and type, defining the component's props interface using the `LocationType`, and then returning a JSX element with the location details.

The Page

We mentioned in “Overview of the User Interface” on page 215 that a location's detail page should be available at the dynamic URL `location/:location_id`. To implement this, create the `location` folder in the `pages` directory and add the `[locationId].tsx` file containing the code in Listing 14-16.

```

import Head from "next/head";
import type {
  GetServerSideProps,
  GetServerSidePropsContext,
  InferGetServerSidePropsType,
  PreviewData,
  NextPage,
} from "next";
import LocationDetail from "components/location-details";
import dbConnect from "middleware/db-connect";
import { findLocationsById } from "mongoose/locations/services";
import { LocationType } from "mongoose/locations/schema";
import { ParsedUrlQuery } from "querystring";

```

```

const Location: NextPage = (
  props: InferGetServerSidePropsType<typeof getServerSideProps>
) => {
  let location: LocationType = JSON.parse(props.data?.location);
  ❶ let title = `The Food Finder - Details for ${location?.name}`;
  return (
    <div>
      <Head>
        <title>{title}</title>
        <meta
          name="description"
          content={`The Food Finder.
            Details for ${location?.name}`}
        />
      </Head>
      <h1>{location?.name}</h1>
      ❷ <LocationDetail location={location} />
    </div>
  );
};

❸ export const getServerSideProps: GetServerSideProps = async (
  context: GetServerSidePropsContext<ParsedUrlQuery, PreviewData>
) => {
  let locations: LocationType[] | [];
  ❹ let { locationId } = context.query;
  try {
    await dbConnect();
    locations = await findLocationsById([locationId as string]);
    ❺ if (!locations.length) {
      throw new Error(`Locations ${locationId} not found`);
    }
  } catch (err: any) {
    return {
      notFound: true,
    };
  }
  return {
    ❻ props: { data: { location: JSON.stringify(locations.pop()) } },
  };
};

```

Listing 14-16: The pages/location/[locationId].tsx file

The start page and location detail page look fairly similar. The only visual difference is the page's title, which we construct with the location's name ❶, and instead of the `LocationsList` component, we use the `LocationDetail` component with a single location object ❷.

From a functional perspective, however, the pages are not similar. Unlike the start page, which uses SSG, the location detail page uses SSR with `getServerSideProp` ❸. This is because as soon as we add the wish list functionality and implement the Add To/Remove button, the page's

content should change along with a user's action. Hence, we need to regenerate the HTML on each request. We discussed the differences between SSR and SSG in depth in Chapter 5.

We use the page's context and its query property to get the location ID from the dynamic URL ❹. Then we use the ID to get the matching location from the database. As before, we use the service directly instead of calling the publicly exposed API, as Next.js runs both `get...Prop` functions on the server side and can directly access the services in our application's middleware.

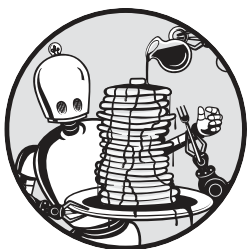
We also implement two exit scenarios. First, if there is no result, we throw an error to step into the catch block ❺, and by doing so, redirect the user to the *404 Not Found* error page. Otherwise, we store the first location from the results in the location property ❻ and pass it to the Next.js page function we export in the last line.

Summary

We've successfully built the frontend for the Food Finder application. At this point, you've implemented a full-stack web application that reads data from a MongoDB database and renders the results as React user interface components in Next.js. Next, we'll add an OAuth authentication flow with GitHub so that users can log in with their GitHub account and store a personalized wish list.

15

ADDING OAUTH



In this chapter, you'll add OAuth authentication to the Food Finder app, giving users the opportunity to log in with their GitHub accounts. You'll also implement the wish list page to which authenticated users can add and remove locations, as well as the button component needed to accomplish this. Lastly, you'll learn how to protect your GraphQL mutations from unauthenticated users.

Adding OAuth with *next-auth*

Developers usually use third-party libraries or SDKs to implement OAuth. For the Food Finder application, we'll use the *next-auth* package from Auth.js, which comes with an extensive set of preconfigured templates that allow us to connect to an OAuth service easily. These templates are called *providers*, and we'll use one of them: the GitHub provider, which adds a Log In with GitHub

button to our app. For a refresher on the OAuth authentication process, return to Chapter 9.

Creating a GitHub OAuth App

First we need to create an OAuth application using GitHub. This should give us the client ID and client secret that the Food Finder application needs to connect to GitHub. If you don't already have a GitHub account, create one now at <https://github.com>, then log in. Navigate to <https://github.com/settings/developers> and create a new OAuth app in the OAuth Apps section. Enter the Food Finder app's details in the resulting form, which should look similar to Figure 15-1.

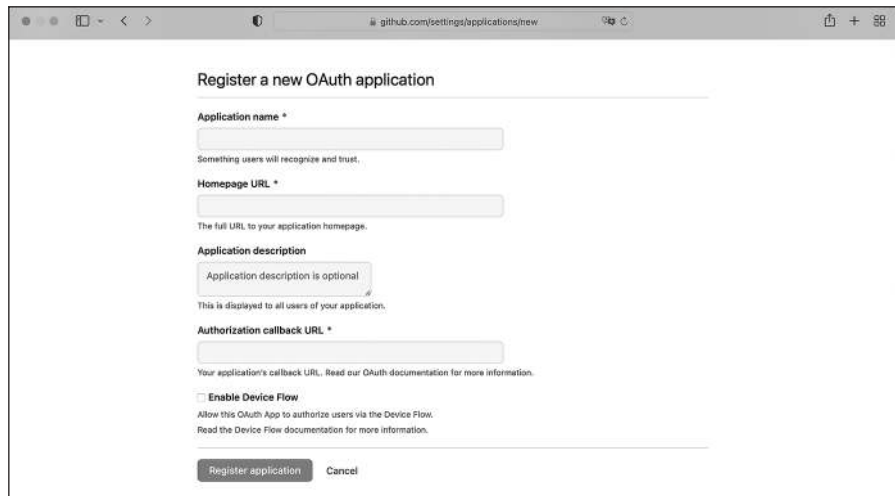
The image shows a web browser window with the URL 'github.com/settings/applications/new'. The page title is 'Register a new OAuth application'. It contains several input fields: 'Application name' with a required asterisk and a hint 'Something users will recognize and trust.'; 'Homepage URL' with a required asterisk and a hint 'The full URL to your application homepage.'; 'Application description' with an optional hint 'Application description is optional' and a note 'This is displayed to all users of your application.'; and 'Authorization callback URL' with a required asterisk and a hint 'Your application's callback URL. Read our OAuth documentation for more information.' Below these fields is a checkbox for 'Enable Device Flow' with a hint 'Allow this OAuth App to authorize users via the Device Flow. Read the Device Flow documentation for more information.' At the bottom are two buttons: 'Register application' and 'Cancel'.

Figure 15-1: The GitHub user interface for adding a new OAuth application

Enter **Food Finder** as the name, set the home page URL to **http://localhost:3000/**, and set the authorization callback URL to **http://localhost:3000/api/auth/callback/github**. After registering the application, GitHub should show us the client ID and let us generate a client secret.

Adding the Client Credentials

Now copy these credentials as `GITHUB_CLIENT_ID` and `GITHUB_CLIENT_SECRET` to the `env.local` file in the application's code root folder. This file looks like this:

```
MONGO_URI=mongodb://backend:27017/foodfinder
GITHUB_CLIENT_ID=ADD_YOUR_CLIENT_ID_HERE
GITHUB_CLIENT_SECRET=ADD_YOUR_CLIENT_SECRET_HERE
```

Fill in the placeholders with your credentials.

Installing next-auth

To add Auth.js's OAuth SDK for *next-auth* to the Food Finder app and configure it to connect to the provider, run the following:

```
$ docker exec -it foodfinder_application npm install next-auth
```

By default, this SDK uses encrypted JWTs to store and attach session information to API requests. The library automatically handles the encryption and decryption as long as we provide it with a secret. To add such a secret, open the *env.local* file and add the following line to the end:

```
NEXTAUTH_SECRET=78f6cc4bf633b1102f4ca4d72602c60f
```

Use any secret you'd like. The string used here was randomly generated with *OpenSSL* at <https://www.usemodernfullstack.dev/api/v1/generate-secret>, and you should use a fresh one for each application.

Creating the Authentication Callback

Now we'll develop the *api/auth* route for the authorization callback URL we supplied to GitHub when registering the OAuth application. Create the *auth* folder in the *pages/api* directory containing the file *[...nextauth].ts*. The ... in the filename tells the Next.js router that this is a catch all, meaning it handles all API calls to endpoints below */auth*; for example, *auth/signin* or *auth/callback/github*. Add the code from Listing 15-1 to the file.

```
import GithubProvider from "next-auth/providers/github";
import { NextApiRequest, NextApiResponse } from "next";
import NextAuth from "next-auth";
import { createHash } from "crypto";

const createUserId = (base: string): string => {
  return createHash("sha256").update(base).digest("hex");
};

export default async function auth(req: NextApiRequest, res: NextApiResponse) {
  return await NextAuth(req, res, {
    providers: [
      GithubProvider({
        clientId: process.env.GITHUB_CLIENT_ID || "",
        clientSecret: process.env.GITHUB_CLIENT_SECRET || "",
      }),
    ],
    callbacks: {
      async jwt({ token }) {
        if (token?.email && !token.fdlst_private_userId) {
          token.fdlst_private_userId = createUserId(token.email);
        }
        return token;
      },
    },
  });
}
```

```

    async session({ session }) {
      if (
        session?.user?.email &&
        !session?.user.fdlst_private_userId
      ) {
        session.user.fdlst_private_userId = createUserId(
          session?.user?.email
        );
      }
      return session;
    },
  },
});
}

```

Listing 15-1: The pages/api/auth/[...nextauth].ts file

We import our dependencies, including the built-in `GithubProvider` and the default `crypto` module. Then we create a simple `createUserId` function, which takes a string as an argument and calls the `crypto` module's `createHash` function to return the hashed user ID from this string.

Next, we create and export the default asynchronous auth function. To do so, we initialize the `NextAuth` module and add the `GithubProvider` to the `providers` array. We configure it to use the `clientId` and the `clientSecret` we stored in the environment variables.

Since we want to keep our application as simple as possible, we'll keep it stateless; hence, we use the `jwt` and `session` callbacks, which *next-auth* uses every time it creates a new session or JWT internally. In the callback, we calculate the hashed user ID from the user's email with our `createId` function (if it's not already available in the current token or session object). Finally, we store it in a private claim.

We've just created a new property, `fdlst_private_userId`, on the user object in the *next-auth* session. As expected, TSC warns us that this property doesn't exist on the `Session` type. We need to augment the type's interface by adjusting the *customs.d.ts* file in our application's root directory to match Listing 15-2.

```

import mongoose from "mongoose";
import { DefaultSession } from "next-auth";

declare global {
  var mongoose: mongoose;
}

declare module "next-auth" {
  interface Session {
    user: {
      fdlst_private_userId: string;
    } & DefaultSession["user"];
  }
}

```

Listing 15-2: The updated customs.d.ts file with the augmented Session interface

In the updated code, we import the *next-auth* package's `DefaultSession`, which defines the default session object, and then create and redeclare the `Session` interface's user object with the new `fdlst_private_userId` property. Because TypeScript overwrites the existing user object, we explicitly add it from the `DefaultSession` object. In other words, we add our new `fdlst_private_userId` property to the `Session` interface.

Sharing the Session Across Pages and Components

With the callback URL set up, we need to ensure that a user's session is shared among all Next.js pages and React components. We can use the `useContext` hook discussed in Chapter 4, which *next-auth* provides for us. In the `pages/_app.tsx` file, wrap the application in a `SessionProvider`, as shown in Listing 15-3.

```
import "../styles/globals.css";
import "../styles/layout.css";
import type { AppProps } from "next/app";
import Layout from "components/layout";
import { SessionProvider } from "next-auth/react";

export default function App({
  Component, pageProps: { session, ...pageProps } }: AppProps) {
  return (
    <SessionProvider session={session}>
      <Layout>
        <Component {...pageProps} />
      </Layout>
    </SessionProvider>
  );
}
```

Listing 15-3: The modified `pages/_app.tsx` file

We import the `SessionProvider` from the *next-auth* package and enhance the `pageProps` with the session object. We store the current session in the provider's session attribute, making it available throughout the Next.js application.

Before we can access the session in the frontend and middleware, we need to add the auth-element with the Sign In button, which will allow users to log in.

The Generic Button Component

It's time to implement the generic button component we mentioned earlier. Technically, this component will be a generic `div` element that we'll style to look like a button, with a few variations. It will serve as the Sign In/Sign Out button in the auth-element and the Add To/Remove From button in the location detail component. Create a new folder, *button*, in the *components* folder, adding an *index.module.css* file with the code in Listing 15-4, as well as an *index.tsx* file.

```
.root {
  align-items: center;
  border-radius: 5px;
```

```

        color: #1d1f21;
        cursor: pointer;
        display: inline-flex;
        font-weight: 500;
        height: 35px;
        letter-spacing: 0;
        margin: 0;
        overflow: hidden;
        place-content: flex-start;
        position: relative;
        white-space: nowrap;
    }

    .root > a,
    .root > span {
        padding: 0 1rem;
        white-space: nowrap;
    }

    .root {
        transition: border-color 0.25s ease-in, background-color 0.25s ease-in,
            color 0.25s ease-in;
        will-change: border-color, background-color, color;
    }

    .root.default,
    .root.default:link,
    .root.default:visited {
        background-color: transparent;
        border: 1px solid transparent;
        color: #1d1f21;
    }

    .root.default:hover,
    .root.default:active {
        background-color: transparent;
        border: 1px solid #dbd8e3;
        color: #1d1f21;
    }

    .root.blue,
    .root.blue:link,
    .root.blue:visited {
        background-color: rgba(0, 118, 255, 0.9);
        border: 1px solid rgba(0, 118, 255, 0.9);
        color: #fff;
        text-decoration: none;
    }

    .root.blue:hover,
    .root.blue:active {
        background-color: transparent;
        border: 1px solid #1d1f21;
        color: #1d1f21;
    }

```

```

    text-decoration: none;
  }

  .root.outline,
  .root.outline:link,
  .root.outline:visited {
    background-color: transparent;
    border: 1px solid #dbd8e3;
    color: #1d1f21;
    text-decoration: none;
  }

  .root.outline:hover,
  .root.outline:active {
    background-color: transparent;
    border: 1px solid rgba(0, 118, 255, 0.9);
    color: rgba(0, 118, 255, 0.9);
    text-decoration: none;
  }

  .root.disabled,
  .root.disabled:link,
  .root.disabled:visited {
    background-color: transparent;
    border: 1px solid #dbd8e3;
    color: #dbd8e3;
    text-decoration: none;
  }

  .root.disabled:hover,
  .root.disabled:active {
    background-color: transparent;
    border: 1px solid #dbd8e3;
    color: #dbd8e3;
    text-decoration: none;
  }

```

Listing 15-4: The components/button/index.module.css file

We add styles for each of the button variations we'd like to create. All are 35 pixels tall and have rounded corners. We define a default style, a variation with a blue background and white color, and an outlined version whose background is white. In addition, we define styles to use for deactivated buttons.

With the styles in place, we can write code for the component. Copy the contents of Listing 15-5 into the component's *index.tsx* file.

```

import React from "react";
import styles from "../index.module.css";

interface PropsInterface {
  disabled?: boolean;
  children?: React.ReactNode;
  variant?: "blue" | "outline";
  clickHandler?: () => any;
}

```

```

const Button = (props: PropsInterface): JSX.Element => {
  const { children, variant, disabled, clickHandler } = props;

  const renderContent = (children: React.ReactNode) => {
    if (disabled) {
      return (
        <span className={styles.span}>
          {children}
        </span>
      );
    } else {
      return (
        <span className={styles.span} onClick={clickHandler}>
          {children}
        </span>
      );
    }
  };

  return (
    <div
      className={[
        styles.root,
        disabled ? styles.disabled : "",
        styles[variant || "default"],
      ].join(" ")}
    >
      {renderContent(children)}
    </div>
  );
};

export default Button;

```

Listing 15-5: The components/button/index.tsx file

After importing the dependencies, we define the interface for the component's prop argument. We also define the `Button` component as a function that returns a JSX element and then use object-destructuring syntax to split the props object into constants representing the object's key-value pairs. We define the internal `renderContent` function with one argument, `children`, typed as a `ReactNode` and rendered wrapped in a `span` element. Depending on the state of the `disabled` property, we also add the click handler from the props object. The component itself returns a `div` that we styled to look like a button.

The AuthElement Component

Although we've added the *next-auth* package to the project, created the OAuth API route, and configured our OAuth provider, we still can't access session information, as there is no Sign In button. Let's create this `AuthElement` component and then add it to the header. This component uses our default

button component, and as soon as the user is logged in, it displays their full name, as well as a link to their wish list.

Create the folder *auth-element* in the *components/header* directory and then add the *index.module.css* file with the code in Listing 15-6.

```
.root {
  align-items: center;
  display: flex;
  justify-content: space-between;
  margin: 0;
  padding: 1rem 0;
  width: auto;
}

.root > * {
  margin: 0 0 0 2rem;
}

.name {
  margin: 1rem 0 0 0;
}

@media (min-width: 600px) {
  .name {
    margin: 0 0 0 1rem;
  }
}
```

Listing 15-6: The components/header/auth-element/index.module.css file

We define a set of basic styles for the component, using a flexbox and margins to align them vertically, and change their layout for smaller screens.

To write the component itself, add an *index.tsx* file to the *auth-element* folder and enter the code from Listing 15-7 into it.

```
import Link from "next/link";
import { signIn, signOut, useSession } from "next-auth/react";
import Button from "components/button";
import styles from "./index.module.css";

const AuthElement = (): JSX.Element => {
  const { data: session, status } = useSession();

  return (
    <>
      { status === "authenticated" (
        <span className={styles.name}>
          Hi <b>{session?.user?.name}</b>
        </span>
      )}
    </>
  )
}
```

```

    <nav className={styles.root}>
      {status === "authenticated" && (
        <>
          <Button variant="outline">
            <Link
href={` /list/${session?.user.fdlst_private_userId}`}
              >
                Your wish list
            </Link>
          </Button>

          <Button variant="blue" clickHandler={() => signOut()}>
            Sign out
          </Button>
        </>
      )}
      {status == "unauthenticated" && (
        <>
          <Button variant="blue" clickHandler={() => signIn()}>
            Sign in
          </Button>
        </>
      )}
    </nav>
  </>
);
};
export default AuthElement;

```

Listing 15-7: The components/header/auth-element/index.tsx file

The most notable imports are the `signIn` and `signOut` functions and the `useSession` hook from *next-auth*. The latter enables us to access session information easily, whereas the two functions trigger the sign-in flow or terminate the session.

We then define the `AuthElement` component and retrieve the session data and the session status from the `useSession` hook. We need both of these to construct the JSX element we return from the component. On the client side, we can access the session information directly via the `useSession` hook. On the server side, though, we'll need to access it through the JWT, because the session information is part of the API request's HTTP cookies.

When the session's status is authenticated, we render the user's name from the session data and add the Your Wish List and Sign Out buttons to the navigation's `nav` element. Otherwise, we add the Sign In button to start the OAuth flow. For all of those, we use the generic button component and the `signIn` and `signOut` functions we imported from the *next-auth* module, both of which handle the OAuth flow automatically.

We use the `next/link` element to link to the user's wish list. (This is another Next.js page we'll implement in a moment.) The wish list is available at the dynamic route `/list/:userId`, which uses the user ID we created by hashing the user's email address and storing it in `fdlst_private_userId`.

Adding the AuthElement Component to the Header

Now we have to add the new component to the header. Open the *index.tsx* file in the *components/header* directory and adjust it so that it matches Listing 15-8.

```
import styles from "../index.module.css";
import Logo from "components/header/logo";
import AuthElement from "components/header/auth-element";
const Header = (): JSX.Element => {
  return (
    <header className={styles.root}>
      <div className="layout-grid">
        <Logo />
        <AuthElement />
      </div>
    </header>
  );
};

export default Header;
```

Listing 15-8: The modified components/header/index.tsx file

The update is simple; we import the *AuthElement* component and add it next to the *Logo* inside the header.

Test the OAuth workflow to see our session management in practice. When you open *http://localhost:3000*, the Sign In button should be in the header, as in Figure 15-2.

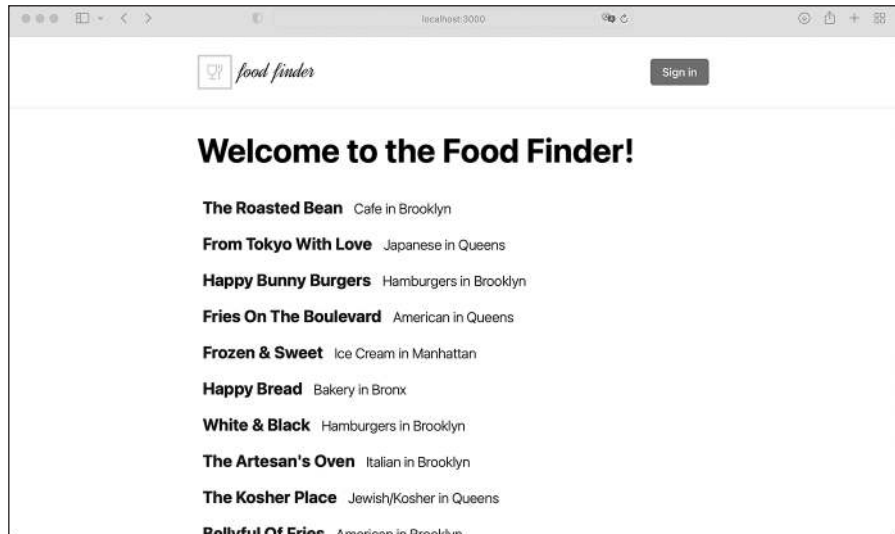


Figure 15-2: The header in a logged-out state with the Sign In button

Let's log in using OAuth. Click the **Sign In** button, and *next-auth* should redirect you to the login screen, where you can select to sign in with the configured OAuth providers to use (Figure 15-3).

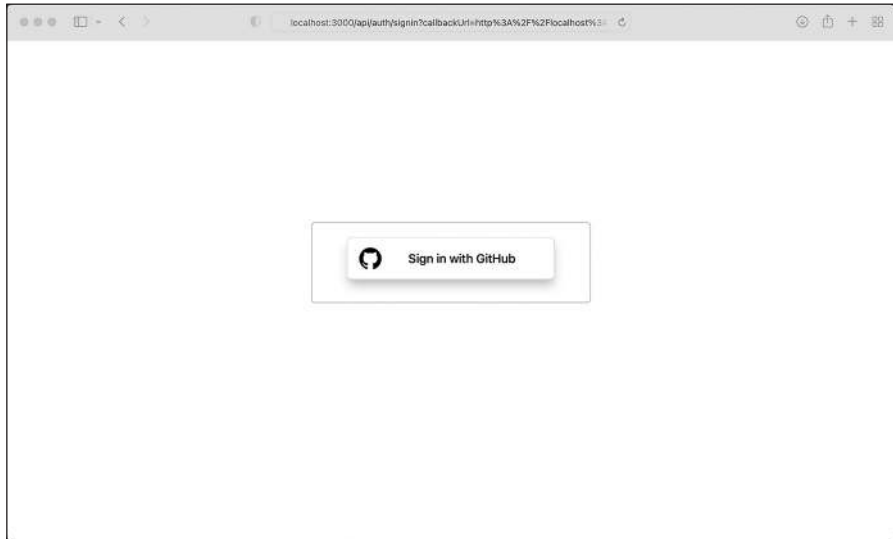


Figure 15-3: OAuth requires us to choose a provider.

Click the button to log in. OAuth should redirect you to the application, where the `AuthElement` renders your name and new buttons based on the session information. The screen should look similar to Figure 15-4.

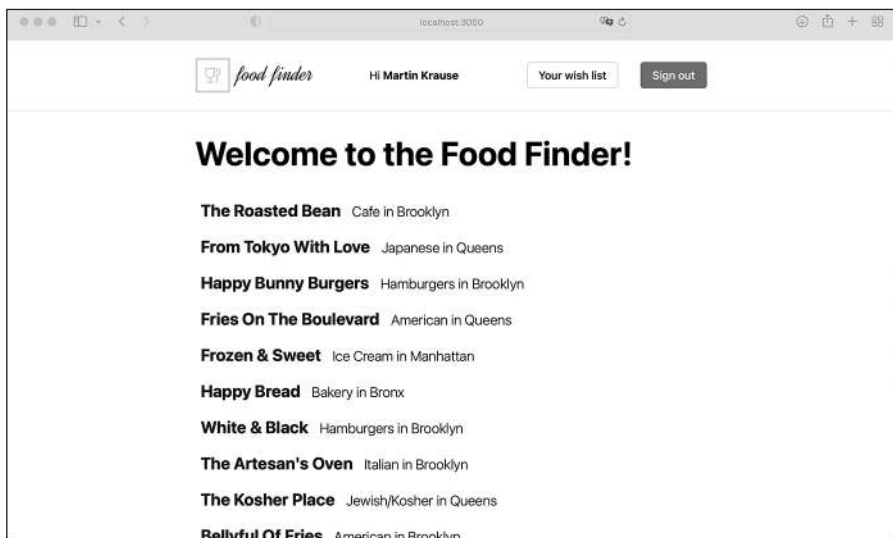


Figure 15-4: The header in the logged-in state with the session information

The header element has changed according to the session's state. We display the current user's name received from the OAuth provider, the link to their public wish list, and the Sign Out button.

The Wish List Next.js Page

The wish list button in the header should link to a wish list page at the dynamic URL `list/:userId`. This regular Next.js page should display all locations whose `on_wishlist` property contains the user ID specified in the dynamic URL. It will look quite similar to the start page, and we can build it out of existing components.

To create the page's route, create the `list` folder with the `[userId].tsx` file in the `pages` directory. Then add the code from Listing 15-9 to this `.tsx` file.

```
import type {
  GetServerSideProps,
  GetServerSidePropsContext,
  NextPage,
  PreviewData,
  InferGetServerSidePropsType,
} from "next";
import Head from "next/head";
import { ParsedUrlQuery } from "querystring";

import dbConnect from "middleware/db-connect";
import { onUserWishlist } from "mongoose/locations/services";
import { LocationType } from "mongoose/locations/schema";
import LocationsList from "components/locations-list";

import { useSession } from "next-auth/react";

const List: NextPage = (
  props: InferGetServerSidePropsType<typeof getServerSideProps>
) => {
  const locations: LocationType[] = JSON.parse(props.data?.locations);
  const userId: string | undefined = props.data?.userId;
  const { data: session } = useSession();
  let title = `The Food Finder- A personal wish list`;
  let isCurrentUsers =
    userId && session?.user.fdlst_private_userId === userId;
  return (
    <div>
      <Head>
        <title>{title}</title>
        <meta content={`The Food Finder. A personal wish list.`} />
      </Head>
      <h1>
        {isCurrentUsers ? " Your " : " A "}
        wish list!
      </h1>
      {isCurrentUsers && locations?.length === 0 && (
        <>
```

```

        <h2>Your list is currently empty! :(</h2>
        <p>Start adding locations to your wish list!</p>
      </>
    )}
    <LocationsList locations={locations} />
  </div>
);
};

export const getServerSideProps: GetServerSideProps = async (
  context: GetServerSidePropsContext<ParsedUrlQuery, PreviewData>
) => {
  let { userId } = context.query;
  let locations: LocationType[] | [] = [];
  try {
    await dbConnect();
    locations = await onUserWishlist(userId as string);
  } catch (err: any) {}
  return {
    // the props will be received by the page component
    props: {
      data: { locations: JSON.stringify(locations), userId: userId },
    },
  };
};
export default List;

```

Listing 15-9: The pages/list/[userId].tsx file

Although we want the wish list page to look similar to the start page, we use SSR, with `getServerSideProps`, as we did for the location detail page. The wish list page is highly dynamic; hence, we need to regenerate the HTML on each request.

Another approach would be to use client-side rendering, then request the user's locations through the GraphQL API in a `useEffect` hook. However, this would cause the user to see a loading screen each time they opened the wish list page. We can avoid this inferior user experience altogether with SSR.

In the server-side part of the page's code, we first extract the URL parameter, `userId`, from the context's query object. We use the user's ID and the `onUsersWishlist` service to get all locations for the user's wish list. If there is an error, we simply continue instead of redirecting to the `404` error page, rendering an empty list.

We then pass the locations array and the user's ID to the Next.js page, where we extract the locations as usual, as well as the `userId`. We compare the user ID from the URL with the user ID in the current session. If they match, we know that the currently logged-in user has visited their own wish list and adjust the user interface accordingly.

Adding the Button to the Location Detail Component

We can now visit the wish list page, but it will always be empty. We haven't yet provided users with a way to add items to it. To change this, we'll place

a button in the location details component that lets users add or remove a particular location. We'll use the generic button component and session information. Open the *index.ts* file in the *components/location-details.tsx* directory and modify the code to match Listing 15-10.

```
import { LocationType } from "mongoose/locations/schema";
import styles from "../index.module.css";

import { useSession } from "next-auth/react";
import { useEffect, useState } from "react";
import Button from "components/button";

interface PropsInterface {
  location: LocationType;
}

interface WishlistInterface {
  locationId: string;
  userId: string;
}

const LocationDetail = (props: PropsInterface): JSX.Element => {
  let location: LocationType = props.location;

  const { data: session } = useSession();
  const [onWishlist, setOnWishlist] = useState<Boolean>(false);
  const [loading, setLoading] = useState<Boolean>(false);

  useEffect(() => {
    let userId = session?.user.fdlst_private_userId;
    setOnWishlist(
      userId && location.on_wishlist.includes(userId) ? true : false
    );
  }, [session]);

  const wishlistAction = (props: WishlistInterface) => {

    const { locationId, userId } = props;

    if (loading) { return false; }
    setLoading(true);

    let action = !onWishlist ? "addWishlist" : "removeWishlist";

    fetch("/api/graphql", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        query: `mutation wishlist {
          ${action}({
            location_id: "${locationId}",
            user_id: "${userId}"
          }
        }`
      })
    })
  }
}
```

```

        ) {
            on_wishlist
        }
    },
    }},
    })
    .then((result) => {
        if (result.status === 200) {
            setOnWishlist(action === "addWishlist" ? true : false);
        }
    })
    .finally(() => {
        setLoading(false);
    });
};

return (
    <div>
        {location && (
            <ul className={styles.root}>
                <li>
                    <b>Address: </b>
                    {location.address}
                </li>
                <li>
                    <b>Zipcode: </b>
                    {location.zipcode}
                </li>
                <li>
                    <b>Borough: </b>
                    {location.borough}
                </li>
                <li>
                    <b>Cuisine: </b>
                    {location.cuisine}
                </li>
                <li>
                    <b>Grade: </b>
                    {location.grade}
                </li>
            </ul>
        )}

        {session?.user.fdlst_private_userId && (
            <Button
                variant={!onWishlist ? "outline" : "blue"}
                disabled={loading ? true : false}
                clickHandler={() =>
                    wishlistAction({
                        locationId: session?.user.fdlst_private_userId,
                        userId: session?.user?.userId,
                    })
                }
            />
        )}
    </div>
);

```



```

        >
        {onWishlist && <>Remove from your Wishlist</>}
        {!onWishlist && <>Add to your Wishlist</>}
      </Button>
    )}
  </div>
);
};
export default LocationDetail;

```

Listing 15-10: The modified components/location-details/index.tsx file

First we import `useSession` from `next-auth`, `useEffect` and `useState` from `React`, and the generic `Button` component. Then we define `WishlistInterface`, the interface for the `wishlistAction` function we'll implement in a bit.

Inside the component, we get the session from the `useSession` hook, then create the `onWishlist` and loading state variables with `useState` as Boolean values. We use the first state variable to specify whether a location is currently on the user's wish list, then update the user interface accordingly. We calculate the initial state in the `useEffect` hook based on the location's `on_wishlist` property. As soon as we've successfully added or removed the location to or from the wish list, we update the state variable and the button's text.

We implement the `wishlistAction` function to update the `on_wishlist` property. First we deconstruct the argument object and then check the loading state to see if there is currently a running request. If so, we exit the function. Otherwise, we set the loading state to true to block the user interface, calculate the action for the GraphQL mutations, and use it to call the `wishlist` mutation. After successfully modifying the document in the database, we update the `onWishlist` state and unblock the user interface.

We check the current session to see if the user is logged in. If so, we render the `Button` component and set the `disabled` and `class name` attributes based on the loading state, as well as an `on-click` event. With each click of the button, we call the `wishlistAction` function with the current location ID and user ID as arguments. Finally, we set the button's text based on the `onWishlist` state, either adding the current location to the wish list or removing it.

Try adding and removing a few locations to the wish list before moving on. Check that the button's text changes accordingly and that a list of locations similar to the one on the start page appears on the wish list page.

Securing the GraphQL Mutations

There is one more thing we have to do to wrap up the application: secure the GraphQL API. While the queries should be publicly available, the mutations should be accessible only to logged-in users, who should be able to add or remove only their own user ID for the `on_wishlist` property.

But if you test the API with the `curl` command, you'll see that, currently, everyone can access the API. Note that you must enter the values supplied to the `-d` flag on a single line, or the server might return an error:

```
$ curl -v \
  -X POST \
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
  -d '{"query":"mutation wishlist {removeWishlist(location_id: \"12340\",
user_id: \"exampleid\") {on_wishlist}}\"}' \
  http://localhost:3000/api/graphql

< HTTP/1.1 200 OK
<
{"data":{"removeWishlist":{"on_wishlist":[]}}}
```

As a test, we send a simple mutation to remove the location with the ID 12340 from a nonexistent user's wish list. (The mutation won't work, which is fine; we just want to verify whether the API is accessible to the public.) The command receives a *200* response and the expected JSON, proving that the mutations are public.

Let's implement an `authGuard` to protect our mutations. A *guard* is a pattern that checks a condition and then throws an error if it isn't met, and an *auth guard* protects a route or an API from unauthorized access.

We begin by creating the file `auth-guards.ts` in the `middleware` folder and adding the code in Listing 15-11.

```
import { GraphQLError } from "graphql/error";
import { JWT } from "next-auth/jwt";

interface paramInterface {
  user_id: string;
  location_id: string;
}

interface contextInterface {
  token: JWT;
}

export const authGuard = (
  param: paramInterface,
  context: contextInterface
): boolean | Error => {
  ❶ if (!context || !context.token || !context.token.fdlst_private_userId) {
    return new GraphQLError("User is not authenticated", {
      extensions: {
        http: { status: 500 },
        code: "UNAUTHENTICATED",
      },
    });
  }
}
```

```

❷ if (context?.token?.fdlst_private_userId !== param.user_id) {
    return new GraphQLError("User is not authorized", {
        extensions: {
            http: { status: 500 },
            code: "UNAUTHORIZED",
        },
    });
}
return true;
};

```

Listing 15-11: The middleware/auth-guards.ts file

We also import JWT from next-auth and the GraphQLError constructor from graphql. We'll use the latter to create the error objects returned to the user if authentication fails. Next, we define our interfaces for the authGuard function's arguments and export the function itself.

We'll call the auth guard from the mutation resolver with two parameters: an object with the user ID and the location ID, for which we defined the paramInterface, and the context object with the token, the contextInterface. The auth guard returns either a Boolean indicating that authentication succeeded or an error. In the authGuard function, we verify that every access to our mutation has a token with a private claim ❶ and that the user ID in the private claim matches the user ID we pass to the mutation ❷. In other words, we verify that a logged-in user has made the API request and that they're modifying their own wish list.

If the checks fail, we create an error with a message and code. In addition, we set the HTTP status code to 500. Remember that unlike REST APIs, which rely on an extensive list of precise HTTP status codes to communicate with the caller, a GraphQL API usually uses either 200 or 500 as the status code for errors. Broadly speaking, we send a 500 status code when GraphQL can't execute the query at all and 200 when the query can be executed. In both cases, the GraphQL API should include precise information about what error occurred.

Now we must pass the user's OAuth token to the resolvers, which will then pass it to the auth guard. To do so, we'll use the context function we implemented in the startServerAndCreateNextHandler function, found in the *pages/api/graphql.ts* file. Open the file and adjust it to match the code in Listing 15-12.

```

import { ApolloServer, BaseContext } from "@apollo/server";
import { startServerAndCreateNextHandler } from "@as-integrations/next";

import { resolvers } from "graphql/resolvers";
import { typeDefs } from "graphql/schema";
import dbConnect from "middleware/db-connect";

import { NextApiHandler, NextApiRequest, NextApiResponse } from "next";

import { getToken } from "next-auth/jwt";

```

```

const server = new ApolloServer<BaseContext>({
  resolvers,
  typeDefs,
});

const handler = startServerAndCreateNextHandler(server, {
  context: async (req: NextApiRequest) => {
    const token = await getToken({ req });
    return { token };
  },
});

const allowCors =
  (fn: NextApiHandler) =>
  async (req: NextApiRequest, res: NextApiResponse) => {
    res.setHeader("Allow", "POST");
    res.setHeader("Access-Control-Allow-Origin", "*");
    res.setHeader("Access-Control-Allow-Methods", "POST");
    res.setHeader("Access-Control-Allow-Headers", "*");
    res.setHeader("Access-Control-Allow-Credentials", "true");

    if (req.method === "OPTIONS") {
      res.status(200).end();
    }
    return await fn(req, res);
  };

const connectDB =
  (fn: NextApiHandler) =>
  async (req: NextApiRequest, res: NextApiResponse) => {
    await dbConnect();
    return await fn(req, res);
  };

export default connectDB(allowCors(handler));

```

Listing 15-12: The modified pages/api/graphql.ts file with the JWT token

Unlike on the client side, where we can access the session information directly via the `useSession` hook, here we need to access it through the JWT on the server side. This is because the session information is part of the API request's HTTP cookies on the server instead of the `SessionProvider`'s shared session state, and we need to extract it from the request. To do so, we import the `getToken` function from the `next-auth` `jwt` module. Then we pass the request object we receive from the context function to call `getToken` and await the decoded JWT. Next, we return the token from the context function so that we can access the token in the resolver functions.

Finally, let's use the token to add the `authGuard` to our resolvers to protect them from unauthenticated and unauthorized access. Open the `graphql/locations/mutations.ts` file and update it with the code from Listing 15-13.

```

import { updateWishlist } from "mongoose/locations/services";
import { authGuard } from "middleware/auth-guard";
import { JWT } from "next-auth/jwt";

interface UpdateWishlistInterface {
  user_id: string;
  location_id: string;
}

interface contextInterface {
  token: JWT;
}

export const locationMutations = {
  removeWishlist: async (
    _: any,
    param: UpdateWishlistInterface,
    context: contextInterface
  ) => {

    const guard = authGuard(param, context);
    if (guard !== true) { return guard; }

    return await updateWishlist(param.location_id, param.user_id, "remove");
  },

  addWishlist: async (
    _: any,
    param: UpdateWishlistInterface,
    context: contextInterface
  ) => {

    const guard = authGuard(param, context);
    if (guard !== true) { return guard; }

    return await updateWishlist(param.location_id, param.user_id, "add");
  },
};

```

Listing 15-13: The graphql/locations/mutations.ts file with the added authGuard

We define a new interface for the context and update the context parameter to contain the JWT. Next, we add the `authGuard` function to our mutations and follow the guard pattern by returning the error immediately instead of proceeding with the code.

To test the `authGuard` functionality, run `curl` again. The command line output should look similar to Listing 15-14.

```

$ curl -v \
  -X POST \
  -H "Accept: application/json" \
  -H "Content-Type: application/json" \
  -d '{"query":"mutation wishlist {removeWishlist(location_id: \"12340\"},

```

```

user_id: \"exampleid\") {on_wishlist}}}' \
http://localhost:3000/api/graphql

< HTTP/1.1 500 Internal Server Error
<
{
  "errors": [
    {
      "message": "User is not authenticated",
      "locations": [ { "line": 1, "column": 20} ],
      "path": [ "removeWishlist" ],
      "extensions": { "code": "UNAUTHENTICATED", "data": null }
    }
  ]
}

```

Listing 15-14: The curl command to test our API

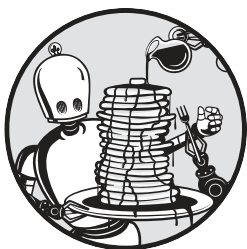
Unlike the previous curl call, the GraphQL API now responds with HTTP/1.1 500 Internal Server Error and an extensive error message, which we defined when we created the GraphQLError in the *auth-guards.ts* file.

Summary

We've successfully added an OAuth authentication flow to the Food Finder application. Now the user can log in with their GitHub account. Once logged in, they can maintain their personal public wish list. In addition, we've protected the GraphQL mutations, meaning they are no longer available to anyone; instead, only logged-in users can access them. In the final chapter, we'll add automated tests to evaluate the application using Jest.

16

RUNNING AUTOMATED TESTS IN DOCKER



In this short final chapter, you'll write a couple of automated tests that verify the state of the Food Finder application. Then you'll configure a Docker service to continuously run them.

We'll focus on evaluating the application's header by using a snapshot test and mocking the user session. We won't create tests for the other components or our middleware, services, or APIs. However, I encourage you to build these on your own. Try using browser-based end-to-end tests, with a specialized framework such as Cypress or Playwright, to test entire pages. You can find installation instructions and examples for both frameworks at <https://nextjs.org/docs/testing>.

Adding Jest to the Project

Install the Jest libraries with npm:

```
$ docker exec -it foodfinder-application npm install --save-dev jest \
jest-environment-jsdom @testing-library/react @testing-library/jest-dom
```

Next, configure Jest to work with our Next.js setup by creating a new file called *jest.config.js* containing the code in Listing 16-1. Save the file in the application's root folder.

```
const nextJest = require("next/jest");

const createJestConfig = nextJest({
  dir: "./",
});

const customJestConfig = {
  moduleDirectories: ["node_modules", "<rootDir>/"],
  testEnvironment: "jest-environment-jsdom",
};

module.exports = createJestConfig(customJestConfig);
```

Listing 16-1: The `jest.config.js` file

We leverage the built-in Next.js Jest configuration, so we need to configure the project's base directory to load the *config* and *.env* files into the test environment. Then we set the location of the module directories and the global test environment. We use a global setting here because our snapshot tests will require a DOM environment.

Now we want to be able to run the tests with npm commands. Therefore, add the two commands in Listing 16-2 to the *scripts* property of the project's *package.json* file.

```
"test": "jest ",
"testWatch": "jest --watchAll"
```

Listing 16-2: Two commands added to the `package.json` file's `scripts` property

The first command executes all available tests once, and the second continuously watches for file changes and then reruns the tests if it detects one.

Setting Up Docker

To run the tests using Docker, add another service to *docker-compose.yml* that uses the Node.js image. On startup, this service will run `npm run testWatch`, the command we just defined. In doing so, we'll continuously run the tests and get instant feedback about the application's state. Modify the file to match the code in Listing 16-3.

```
version: "3.0"
services:

  backend:
    container_name: foodfinder-backend
    image: mongo:latest
    restart: always
    environment:
      DB_NAME: foodfinder
      MONGO_INITDB_DATABASE: foodfinder
    ports:
      - 27017:27017
    volumes:
      - "./.docker/foodfinder-backend/seed-mongodb.js"
      - /docker-entrypoint-initdb.d/seed-mongodb.js
      - mongodb_data_container:/data/db

  application:
    container_name: foodfinder-application
    image: node:lts-alpine
    working_dir: /home/node/code/foodfinder-application
    ports:
      - "3000:3000"
    volumes:
      - ./code:/home/node/code
    depends_on:
      - backend
    environment:
      - HOST=0.0.0.0
      - CHOKIDAR_USEPOLLING=true
      - CHOKIDAR_INTERVAL=100
    tty: true
    command: "npm run dev"

  jest:
    container_name: foodfinder-jest
    image: node:lts-alpine
    working_dir: /home/node/code/foodfinder-application
    volumes:
      - ./code:/home/node/code
    depends_on:
      - backend
      - application
    environment:
      - NODE_ENV=test
    tty: true
    command: "npm run testWatch"

volumes:
  mongodb_data_container:
```

Listing 16-3: The modified docker-compose.yml file with the jest service

Our small service, named *jest*, uses the official Node.js Alpine image we've used previously. We set the working directory and use the volumes

property to make our code available in this container as well. Unlike our application's service, however, the *jest* service sets the Node.js environment to test and runs the *testWatch* command.

Restart the Docker containers; the console should indicate that Jest is watching our files.

Writing Snapshot Tests for the Header Element

As in Chapter 8, create the `__tests__` folder to hold our test files in the application's root directory. Then add the *header.snapshot.test.tsx* file containing the code in Listing 16-4.

```
import { act, render } from "@testing-library/react";
import { useSession } from "next-auth/react";
import Header from "components/header";

jest.mock("next-auth/react");
describe("The Header component", () => {
  it("renders unchanged when logged out", async () => {
    (useSession as jest.Mock).mockReturnValueOnce({
      data: { user: {} },
      status: "unauthenticated",
    });
    let container: HTMLElement | undefined = undefined;
    await act(async () => {
      container = render(<Header />).container;
    });
    expect(container).toMatchSnapshot();
  });

  it("renders unchanged when logged in", async () => {
    (useSession as jest.Mock).mockReturnValueOnce({
      data: {
        user: {
          name: "test user",
          fd1st_private_userId: "rndmusr",
        },
      },
      status: "authenticated",
    });
    let container: HTMLElement | undefined = undefined;
    await act(async () => {
      container = render(<Header />).container;
    });
    expect(container).toMatchSnapshot();
  });
});
```

Listing 16-4: The `__tests__/header.snapshot.test.tsx` file

This test should resemble those you wrote in Chapter 8. Note that we import the *useSession* hook from *next-auth/react* and then use *jest.mock* to

replace it in the *arrange* step of each test. By replacing the session with a mocked one that returns the state, we can verify that the header component behaves as expected for both logged-in and logged-out users. We describe the test suite for the Header component by using the arrange, act, and assert pattern and verify that the rendered component matches the stored snapshot.

The first test case uses an empty session and the *unauthenticated* status to render the header in a logged-out state. The second test case uses a session with minimal data and sets the user's status to *authenticated*. This lets us verify that an existing session shows a different user interface than an empty session does.

If you write additional tests, make sure to add them to the `__tests__` folder.

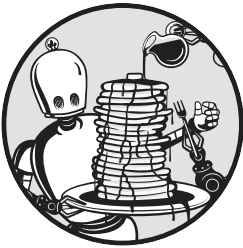
Summary

You've successfully added a few simple snapshot tests to verify that the Food Finder application works as intended. Using an additional Docker service, you can continuously verify that additional developments won't break the application.

Congratulations! You've successfully created your first full-stack application with TypeScript, React, Next.js, Mongoose, and MongoDB. You've used Docker to containerize your application and Jest to test it. With the knowledge gained in the book and its exercises, you've laid the foundation for your career as a full-stack developer.

A

TYPESCRIPT COMPILER OPTIONS



Pass any of these options to the *tsconfig.json* file's *compilerOptions* field to configure TSC's transpilation of TypeScript code to JavaScript. For more information about this process, see Chapter 3.

Here we look at the most common options. You can find more information and the complete list in the official documentation at <https://www.typescriptlang.org/tsconfig>.

allowJs A Boolean that specifies whether the project can import JavaScript files.

baseUrl A string that defines the root directory to use for resolving module paths. For example, if you set it to `"/"`, TypeScript will resolve file imports from the root directory.

esModuleInterop A Boolean that specifies whether TypeScript should import CommonJS, AMD, or UMD modules seamlessly or treat them differently from ES.Next modules. In general, this is necessary if you use third-party libraries without ES.Next module support.

forceConsistentCasingInFileNames A Boolean that specifies whether file imports are case sensitive. This can be important when some developers are working on case-sensitive filesystems and others are not, to ensure file-loading behaviors are consistent for everyone.

incremental A string that defines whether the TypeScript compiler should save the last compilation's project graph, use incremental type checks, and perform incremental updates on consecutive runs. This can make transpiling faster.

isolatedModules A Boolean that specifies whether TypeScript should issue warnings for code not compatible with third-party transpilers (such as Babel). The most common cause for those warnings is that the code uses files that are not modules; for example, they don't have any `import` or `export` statements. This value doesn't change the behavior of the actual JavaScript; it only warns about code that can't be correctly transpiled.

jsx A string that specifies how TypeScript handles JSX. It applies only to `.tsx` files and how the TypeScript compiler emits them; for example, the default value `react` transforms and emits the code by using `React.createElement`, whereas `preserve` does not transform the code in your component and emits it untouched.

lib An array that adds missing features through polyfills. In general, *polyfills* are snippets of code that add support for features and functions the target environment does not support natively. We need to emulate modern JavaScript features when we target less-compliant systems, such as older browsers or node versions. The compiler adds the polyfills defined in the `lib` array to the generated code.

module A string that sets the module syntax for the transpiled code. For example, if you set it to `commonjs`, TSC will transpile this project to use the legacy CommonJS module syntax with `require` for importing and `module.exports` for exporting the code, whereas with `ES2015` the transpiled code will use the `import` and `export` keywords. This is independent of the `target` property, which defines all available language features except the module syntax.

moduleResolution A string that specifies the module resolution strategy. This strategy also defines how TSC locates definition files for modules at compile time. Changing the approach can resolve fringe problems with the importing and exporting of modules.

noEmit A Boolean that defines whether TSC should produce files or only check the types in the project. Set it to `false` if you want third-party tools such as webpack, Babel.js, or Parcel to transpile the code instead of TSC.

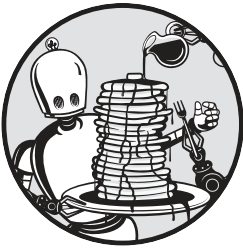
resolveJsonModule A Boolean that specifies whether TypeScript imports JSON files. It generates type definitions based on the JSON inside the file and validates the types on import. We need to manually enable JSON imports as TypeScript can't import them by default.

skipLibCheck A Boolean that defines whether the TypeScript compiler performs type checks on all type declaration files. Setting it to false decreases compilation time and is your escape hatch for working with untyped third-party dependencies.

target A string that specifies the language features to which the TypeScript code should be transpiled. For example, if you set it to es6, or the equivalent ES2015, TSC will transpile this project to ES2015-compatible JavaScript, which, for example, uses let and const.

B

THE NEXT.JS APP DIRECTORY



In version 13, Next.js introduced a new routing paradigm that uses an *app* directory instead of the *pages* directory. This appendix discusses this new feature so that you can explore it further on your own. As there are no plans to deprecate the *pages* directory, you can continue using the routing approach you learned in Chapter 5. You can even use both directories simultaneously; just be careful not to add to both directories folders and files that would create the same route, as this could cause errors.

Both the *app* and *pages* directories use folders and files to create routes. However, the *app* directory distinguishes between server and client components. In the *pages* folder, everything is a *client component*, meaning that all the code is part of the JavaScript bundle Next.js sends to the client. But

every file in the *app* directory is a *server component* by default, and its code is never sent to the client.

This appendix takes a look at the basic concepts of the new approach and then initializes a Next.js application using the new structure.

Server Components vs. Client Components

The terms *client* and *server* in this context refer to the environments in which the Next.js runtime renders a component. The client environment is the user's environment (usually the browser), whereas the server refers to the Next.js server that receives the request from the client, whether it runs on your local host or in a remote location.

With the introduction of server components, Next.js no longer purely uses client-side routing. In *server-centric* routing, the server renders components and then sends the rendered code to the client. This means the client doesn't download a routing map, which reduces the initial page size. Additionally, the user doesn't have to wait until all resources have loaded before the page becomes interactive. Next.js server components leverage React's streaming architecture to progressively render each component's content. With this model, the page becomes interactive before it has finished loading.

Server Components

Next.js server components build upon the React server components that have been available since React version 18. Because the server renders these components, they don't add anything to the JavaScript sent to the client, reducing the overall page size and increasing page performance scores. Also, the JavaScript bundle is cacheable, so the client won't redownload it when we add new additional server components, only when we add new client-side scripts through additional client components.

In addition, because these components are rendered completely on the server, they can contain sensitive server information, such as access tokens and API keys. (To add an additional layer of protection, Next.js's rendering engine replaces with an empty string all environment variables that are not explicitly prefixed with `NEXT_PUBLIC`.) Finally, we can use large-scale dependencies and additional frameworks without bloating the client-side scripts and access backend resources directly, increasing the application's performance.

Listing B-1 shows the basic structure of a server component.

```
export default async function ServerComponent(props: WeatherProps): Promise<JSX.Element> {  
  
  return (  
    <h1>The weather is {props.weather}</h1>  
  );  
}
```

Listing B-1: A basic server component

In Chapter 4, you learned that a React component is a JavaScript function that returns a React element; Next.js server components follow that same structure, except that they're asynchronous functions, so we can use the `async/await` pattern with `fetch`. Thus, instead of returning the React element, it returns a promise of it. The code in Listing B-1 should remind you of the `WeatherComponent` created in the previous chapters, except it doesn't contain any client-side code.

Client Components

By contrast, a client component is a component rendered by the browser rather than by the server. You already know how to write client components, because all React and Next.js components were traditionally client components.

To render these components, the client needs to receive all required scripts and their dependencies. Each component increases the bundle size, decreasing the application's performance. For that reason, Next.js offers options to optimize the application's performance, such as server-side rendering (SSR), which pre-renders the pages on the server, then lets the client add interactive elements to the page.

All components in the `app` directory are server components by default. Client components, however, can reside anywhere (for example, in the `components` directory we've used previously). Listing B-2 shows the `WeatherComponent` created in Listing 5-4 refactored into a client component that works with the `app` directory.

```
"use client";

import React, { useState, useEffect } from "react";

export default function ClientComponent (props: WeatherProps): JSX.Element {

  const [count, setCount] = useState(0);
  useEffect(() => {setCount(1);}, []);

  return (
    <h1
      onClick={() => setCount(count + 1) } >
      The weather is {props.weather},
      and the counter shows {count}
    </h1>
  );
}
```

Listing B-2: A basic client component that is similar to the `WeatherComponent` created in Listing 5-4

We export the component as the default function with the name `ClientComponent`. Because we're using the client-side hooks `useEffect` and `useState` as well as the `onClick` event handler, we need to declare the component as a client component with the `"use client"` directive at the top of the file. Otherwise, Next.js will throw an error.

Rendering Components

In Chapter 5, we performed server-side rendering with the `getServerSideProps` function and used static site generation (SSG) with the `getStaticProps` function. In the `app` directory, both functions are obsolete. If we want to optimize an application, we can instead use Next.js's built-in `fetch` API, which controls data retrieval and rendering at the component level.

Fetching Data

The new asynchronous `fetch` API extends the native `fetch` web API and returns a promise. Because server components are just exported functions that return a JSX element, we can declare them as asynchronous functions and then use `fetch` with the `async/await` pattern.

This pattern is beneficial because it allows us to fetch data for only the segment that uses the data rather than for an entire page. This lets us leverage React features to automatically display loading states and gracefully catch errors, as discussed in “Exploring the Project Structure” on page 269. If we follow this pattern, a loading state will block the rendering of only a particular server component and its user interface; the rest of the page will be fully functional and interactive.

NOTE

Client components shouldn't be asynchronous functions, because the way JavaScript handles asynchronous calls can easily lead to multiple re-renders and slow down the whole application. Next.js developers have discussed adding a generic `use` hook that lets us use asynchronous functions in client components by caching the results, but this hook is not yet finalized. If you absolutely need client-side data fetching, I recommend using a specialized library such as SWR, which you can find at <https://swr.vercel.app>.

You might worry that, when each server component loads its own data, you'll end up with a massive number of requests. How do these numbers impact the overall page performance? Well, Next.js's `fetch` comes with multiple optimizations to speed up the application. For example, it automatically caches the response data for GET requests sent from a server component to the same API, reducing the number of requests.

However, POST requests aren't usually cacheable, as the data they contain might change, so `fetch` won't automatically cache them. This is a problem for us because GraphQL typically uses POST requests. Fortunately, React exposes a `cache` function that memorizes the result of the function it wraps. Listing B-3 shows an example of using `cache` with a GraphQL API.

```
import { cache } from 'react';

export const getUserFromGraphQL = cache(async (id:string) => {
  return await fetch("/graphql," { method: "POST", body: "query:"" });
});
```

Listing B-3: A simple outline of a cached POST API call

We wrap the API call in the cache function we imported from React and return the API's response object. Note that the cached arguments can use only primitive values because the cache function doesn't perform a deep comparison for the arguments.

Another optimization we can implement is to leverage the asynchronous nature of fetch to request data for the server component in a parallel fashion instead of sequentially. Here, the most common pattern is to use `Promise.all` to start all requests at the same time and block the rendering until all requests have been completed. Listing B-4 shows us the relevant code for this pattern.

```
const userPromiseOne = getUserFromGraphQL ("0001");
const userPromiseTwo = getUserFromGraphQL ("0002");

const [userDataOne, userDataTwo] = await Promise.all([userPromiseOne, userPromiseTwo]);
```

Listing B-4: Two parallel API calls with `Promise.all`

We set up two requests, both of which return a promise user object. Then we await the result of both promises and call `Promise.all` with an array of the previously created asynchronous API calls. The `Promise.all` function resolves as soon as both promises return their data, and then the server component's code continues.

Static Rendering

Static rendering is the default setting for both server and client components. It resembles static site generation, which we used with `getStaticProps` in Chapter 5. This rendering option pre-renders both client and server components in the server environment at build time. As a result, requests will always return the same HTML, which remains static and is never re-created.

Each component type is rendered slightly differently. For client components, the server pre-renders the HTML and JSON data; the client then receives the pre-rendered data, including the client-side script, to add interactivity to the HTML. For server components, the browser receives only the rendered payload to hydrate the component. They neither have client-side JavaScript nor use JavaScript for hydration; hence they do not send any JavaScript to the client and, in turn, don't bloat the bundled scripts.

Listing B-5 shows how to statically render the `utils/fetch-names.ts` file from Listing 5-8.

```
export default async function ServerComponentUserList(): Promise<JSX.Element> {
  const url = "https://www.usemodernfullstack.dev/api/v1/users";
  let data: responseItemType[] | [] = [];
  let names: responseItemType[] | [];
  try {
    const response = await fetch(url, { cache: "force-cache" });
    data = (await response.json()) as responseItemType[];
```

```

    } catch (err) {
      throw new Error("Failed to fetch data");
    }
    names = data.map((item) => {
      return { id: item.id, name: item.name };
    });

    return (
      <ul>
        {names.map((item) => (
          <li key="{item.id}">{item.name}</li>
        ))}
      </ul>
    );
  }
}

```

Listing B-5: A server component that uses static rendering

First we define a server component as an asynchronous function that directly returns a `JSX.Element` wrapped in a promise.

In Chapter 5, we returned the page's data and then used the page props to pass it the `NextPage` function, where we generated the element. Here, after setting the `url`, we use the asynchronous `fetch` function to get the data from the remote API. Next.js will cache the results of the API call and the rendered component, and the server will reuse the generated code and never re-create it.

If you use `fetch` without an explicit cache setting, it will use `force-cache` as the default to perform static rendering. To switch to incremental static regeneration instead, replace the `fetch` call from Listing B-5 with the one in Listing B-6.

```
const response = await fetch(url, { next: { revalidate: 20 } });
```

Listing B-6: The modified fetch call for ISR-like rendering

We simply add the `revalidate` property with a value of 30. The server will then render the component statically but invalidate the current HTML 30 seconds after the first page request and re-render it.

Dynamic Rendering

Dynamic rendering replaces Next.js's traditional server-side rendering (SSR), which we used by exporting the `getServerSideProps` function from a page route in Chapter 5. Because Next.js uses static rendering by default, we must actively opt in to using dynamic rendering in one of two ways: by disabling the cache in our fetch requests or by using a dynamic function. In Listing B-7, we disable the cache.

```
export default async function ServerComponentUserList(): Promise<JSX.Element> {
  const url = "https://www.usemodernfullstack.dev/api/v1/users";
  let data: responseItemType[] | [] = [];
  let names: responseItemType[] | [];

```

```

try {
  const response = await fetch(url, { cache: "no-cache" });
  data = (await response.json()) as responseItemType[];
} catch (err) {
  throw new Error("Failed to fetch data");
}
names = data.map((item) => {
  return { id: item.id, name: item.name };
});

return (
  <ul>
    {names.map((item) => (
      <li key="{item.id}">{item.name}</li>
    ))}
  </ul>
);
}

```

Listing B-7: A server component that uses dynamic rendering by disabling the cache

We explicitly set the cache property to no-cache. Now the server will re-fetch the data for the component upon each request.

Instead of disabling the cache, we could use dynamic functions, including the header function or the cookies function in server components and the useSearchParams hook in client components. These functions use dynamic data such as request headers, cookies, and search parameters that are unknown during build time and are part of the request object we pass to the function. The server needs to run these functions for each request because the required data depends on the request.

Keep in mind that dynamic rendering affects the whole route. If one server component in a route opts for dynamic rendering, Next.js will render the whole route dynamically at request time.

Exploring the Project Structure

Let's set up a new Next.js application to explore the features we've discussed. First, use the `npx create-next-app@latest` command with the `--typescript` `--use-npm` flags to create a sample application. When answering the setup wizard's questions, choose to use the *app* directory instead of the *pages* directory.

NOTE

You can also use the online playground at <https://codesandbox.io/s/> to run the Next.js code examples in this appendix. Search for the official Next.js (App router) template when creating a new code sandbox there.

Now enter the `npm run dev` command to start the application in development mode. You should see a Next.js welcome screen in your browser at `http://localhost:3000`. Unlike the welcome screen you saw in Chapter 5,

which encouraged us to edit the *pages/index.tsx* file, here the welcome screen directs us to the *app/page.tsx* file.

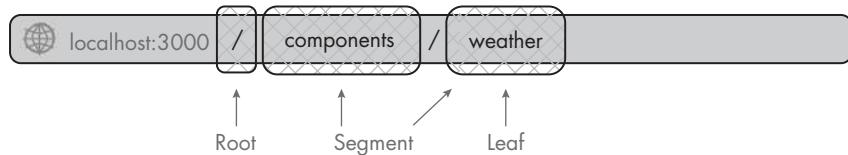
Take a look at the files and folders the wizard created and compare them with the ones from Chapter 5. You should see that the *pages* and *styles* directories are not part of the new structure. Instead, the router replaces both with the *app* directory. Inside it, you should see neither the *_app.tsx* file nor the *_document.tsx* file. Instead, it uses the root layout file *layout.tsx* to define the HTML wrapper for all rendered pages and the *page.tsx* file to render the root segment (the home page).

The *pages* directory uses only one file to create the final content of the page route. By contrast, the *app* directory uses multiple files to create a page route and add additional behavior.

The *page.tsx* file generates the user interface and the content for the route, and its parent folder defines the leaf segment. Without a *page.tsx* file, the URL path won't be accessible. We can then add other special files to the page's folder. Next.js will automatically apply them to this URL segment and its children. The most important of these special files are *layout.tsx*, which creates a general user interface; *loading.tsx*, which uses a React suspense boundary to automatically create a "loading" user interface while the page loads; and *error.tsx*, which uses a React error boundary to catch errors and then show the user a custom error interface.

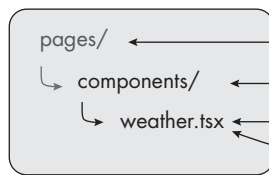
Figure B-1 compares the files and folders for the *components/weather* page route when using the *pages* directory and the *app* directory.

Browser URL:



Files:

Root: *pages* directory



Root: *app* directory

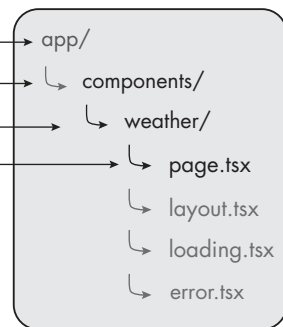


Figure B-1: Comparing the page route components/*weather* in the *pages* and *app* directory structures

When the *app* directory is the root folder, its subfolders still correspond to URL segments, but now the folder that contains the *page.tsx* file defines the URL's final leaf segment. The optional special files next to it affect only the contents of the *components/weather* page.

Let's rebuild the *components/weather* page route you created in Listing 5-1 with the *app* directory. Create the *components* folder and *weather* subfolder inside the *app* directory and then copy the *custom.d.ts* file from the previous code exercises into the root folder.

Updating the CSS

Begin by opening the existing *app/globals.css* file and replacing its content with the code from Listing B-8. We'll need to make some modifications to use special files in our component.

```
html,
body {
  background-color: rgb(230, 230, 230);
  font-family: -apple-system, BlinkMacSystemFont, Segoe UI, Roboto, Oxygen,
    Ubuntu, Cantarell, Fira Sans, Droid Sans, Helvetica Neue, sans-serif;
  margin: 0;
  padding: 0;
}

a {
  color: inherit;
  text-decoration: none;
}

* {
  box-sizing: border-box;
}

nav {
  align-items: center;
  background-color: #fff;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.25);
  display: flex;
  height: 3rem;
  justify-content: space-evenly;
  padding: 0 25%;
}

main {
  display: flex;
  justify-content: center;
}

main .content {
  height: 300px;
  padding-top: 1.5rem;
  width: 400px;
}
```

```

main .content li {
  height: 1.25rem;
  margin: 0.25rem;
}

main .loading {
  animation: 1s loading linear infinite;
  background: #ddd linear-gradient(110deg, #eeeeee 0%, #f5f5f5 15%, #eeeeee 30%);
  background-size: 200% 100%;
  min-height: 1.25rem;
  width: 90%;
}

@keyframes loading {
  to {
    background-position-x: -200%;
  }
}

main .error {
  background: #ff5656;
  color: #fff;
}

section {
  background: #fff;
  border: 1px dashed #888;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.25);
  margin: 2rem;
  padding: 0.5rem;
  position: relative;
}

section .flag {
  background: #888;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.25);
  color: #fff;
  font-family: monospace;
  left: 0;
  padding: 0.25rem;
  position: absolute;
  top: 0;
  white-space: nowrap;
}

```

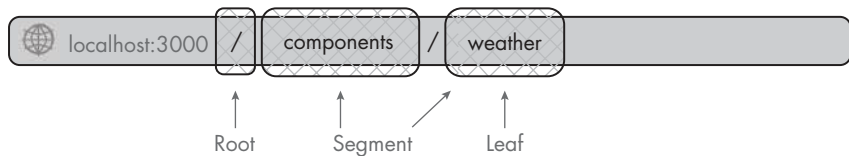
Listing B-8: The app/globals.css file with basic styles for our code examples

We create one `nav` element for the navigation with a main content area below it. Then we add styles for the loading and error states we'll create later. In addition, we use the `section` element to outline the boundaries of the files and `flag` styles to add labels to the sections.

Defining a Layout

Layouts are server components that define the user interface for a particular route segment. Next.js renders this layout when this segment is active. Layouts are shared across all pages, so they can be nested into each other, and all layouts for a specific route and its children will be rendered when this route segment is active. Figure B-2 shows the relationship between the URL, the files, and the component hierarchy for the *components/weather* route.

Browser URL:



Files:

Root: *app* directory

Simplified rendered structure

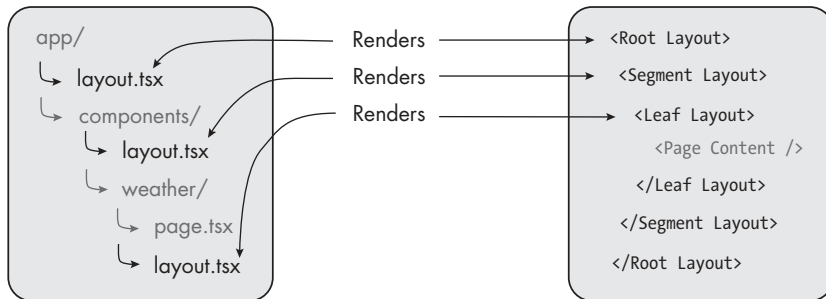


Figure B-2: The simplified layout component hierarchy

In this example, each folder contains a *layout.tsx* file. Next.js will render these in a nested fashion and make the page's content the final rendered component.

Although we can fetch data in a layout, we can't share data between a parent layout and its children. Instead, we can leverage the fetch API's automatic deduplication to reuse data in each child segment or component. When we navigate from one page to another, only the layouts that change are re-rendered. Shared layouts won't be re-rendered when their child segments change.

The root layout, which returns the skeleton structure with the `html` and `body` elements for the page, is required, while all other layouts we create are optional. Let's create a root layout. First, add a new interface to the end of the *custom.d.ts* file, which we copied from the previous exercise. We'll use the `LayoutProps` interface to type the layout's properties object:

```
interface LayoutProps {  
  children: React.ReactNode;  
}
```

Now open the *app/layout.tsx* file and replace its content with the code from Listing B-9.

```
import "../globals.css";

export const metadata = {
  title: "Appendix C",
  description: "The Example Code",
};

export default function RootLayout(props: LayoutProps): JSX.Element {
  return (
    <html lang="en">
      <body>
        <section>
          <span className="flag">app/layout(.tsx)</span>
          {props.children}
        </section>
      </body>
    </html>
  );
}
```

Listing B-9: The file app/layout.tsx defines the root layout.

We import the *global.css* file that we created earlier and then define the default SEO metadata, the page title, and the page description through the metadata object. This replaces the next/head component we used in the *pages* directory for all pages in the *app* directory.

Then we define the *RootLayout* component, which accepts an object of the *LayoutProps* type and returns a *JSX.Element*. We also create the *JSX.Element*, explicitly adding the *html* and *body* elements, then use the *section* and a *span* with the CSS class *flag* to outline the page structure. We add the *children* property from the *LayoutProps* object to wrap them with our root HTML structure.

Now let's add optional layouts to the *app/components* and *app/components/weather* folders. Create a *layout.tsx* file in each and then place the code from Listing B-10 to the *app/components/layout.tsx* file.

```
export default function ComponentsLayout(props: LayoutProps): JSX.Element {
  return (
    <section>
      <span className="flag">app/components/layout(.tsx)</span>
      <nav>Navigation Placeholder</nav>
      <main>{props.children}</main>
    </section>
  );
}
```

Listing B-10: The file app/components/layout.tsx defines the segment layout.

This segment layout file follows the same basic structure as the root layout. We define a layout component that receives the `LayoutProps` object with the `children` property and returns a `JSX.Element`. Unlike in the root layout, we set only the inner structure, the `nav` element with the navigation placeholder, and the main content area where we render the child elements from the `LayoutProps` object, representing this segment's child content (the leaf).

Lastly, create the leaf's layout by adding the code from Listing B-11 to the `app/components/weather/layout.tsx` file.

```
export default function WeatherLayout(props: LayoutProps): JSX.Element {
  return (
    <section>
      <span className="flag">app/components/weather/layout(.tsx)</span>
      {props.children}
    </section>
  );
}
```

Listing B-11: The file `app/components/weather/layout.tsx` defines the leaf layout.

The leaf's layout resembles the segment layout from Listing B-10, but it returns a more straightforward HTML structure, as the `children` property does not contain another layout; instead, it contains the page's content (in `page.tsx`), and the suspense boundary and error boundary from `loading.tsx` and `error.tsx`.

Adding the Content and Route

To expose the page route, we need to create the `page.tsx` file; otherwise, if we tried to visit the `components/weather` page route at `http://localhost:3000/components/weather`, we'd see Next.js's default 404 error page. To re-create the page content from Listing 5-1, we'll create two files. One is `component.tsx`, which contains the `WeatherComponent`, and the other is `page.tsx`, which resembles the `NextPage` wrapper we used in Listing 5-1. Of course, pages could contain additional components located in other folders.

Let's start by creating the `component.tsx` file inside the `apps/components/weather` folder and adding the code from Listing B-12 into it.

```
"use client";

import { useState, useEffect } from "react";

export default function WeatherComponent(props: WeatherProps): JSX.Element {

  const [count, setCount] = useState(0);

  useEffect(() => {
    setCount(1);
  }, []);
```

```

    return (
      <h1 onClick={() => { setCount(count + 1) }} >
        The weather is {props.weather}, and the counter shows {count}
      </h1>
    );
  }

```

Listing B-12: The file `app/components/weather/component.tsx` defines the `WeatherComponent`.

This code is similar to the code in Listing 5-1 for the `WeatherComponent` constant, except we add the "use client" statement to explicitly set it as a client component and export it as the default function instead of storing it in a constant. The component itself has the same functionality as before: we create a headline that shows the weather string and a counter we can increase by clicking the headline.

Now we add the `page.tsx` file and the code from Listing B-13 to create the page route and expose the route to the user.

```

import WeatherComponent from "../component";

export const metadata = {
  title: "Appendix C - The Weather Component (Weather & Count)",
  description: "The Example Code For The Weather Component (Weather & Count)",
};

export default async function WeatherPage() {
  return (
    <section className="content">
      <span className="flag">app/components/weather/page(.tsx)</span>
      <WeatherComponent weather="sunny" />
    </section>
  );
}

```

Listing B-13: The file `app/components/weather/page.tsx` defines the page route.

We import the `WeatherComponent` we just created and then set the SEO metadata on the page level. Then we export the page route as the default async function. When we compare it to Listing 5-1, which contains a similar page, we see that we no longer need to export a `NextPage`; instead, we use a basic function. The `app` directory simplifies the structure of the code.

Now visit our `components/weather` page route at `http://localhost:3000/components/weather` in the browser. You should see a page that looks similar to Figure B-3.

Notice two things here. First, you should recognize the component from Chapter 5, whose counter increases when we click the headline. In addition, the combination of the styles and the `span` elements we added to each `.tsx` file visualizes the relations between the files. We see that the nested layout files resemble the simplified component hierarchy from Figure B-3.

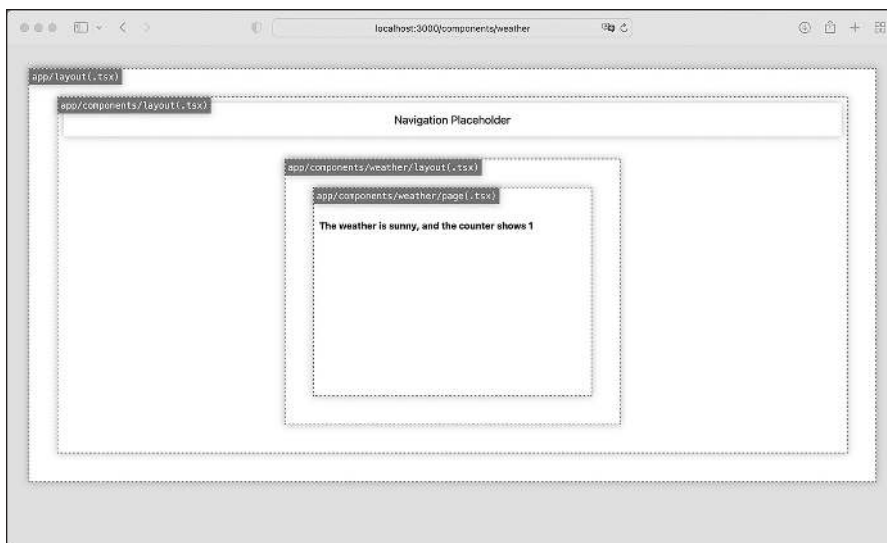
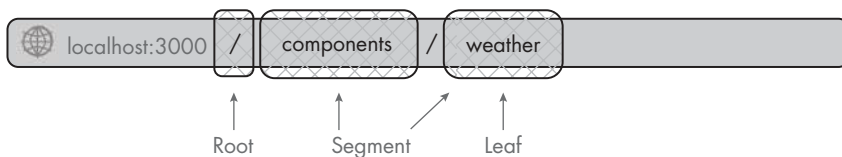


Figure B-3: The `components/weather` page showing the nested components

Catching Errors

As soon as we add an `error.tsx` file to the folder, Next.js wraps our page's content with a React error boundary. Figure B-4 shows the simplified component hierarchy of the `components/weather` route with an added `error.tsx` file.

Browser URL:



Files:

Root: `app` directory

Simplified rendered structure

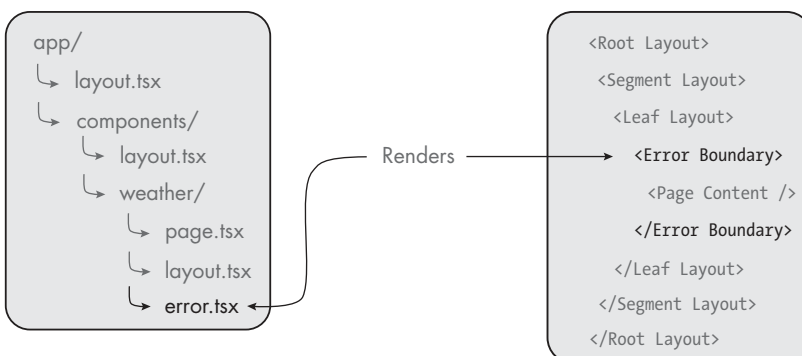


Figure B-4: The simplified layout component hierarchy includes the error boundary.

We see that the *error.tsx* file automatically creates an error boundary around the page's content. By doing so, Next.js enables us to catch errors on a page level and gracefully handle those instead of freezing the whole user interface or redirecting the user to a generic error page. Think about it as a try...catch block on a component level. We can now show a tailored error message and display a button that lets the user re-render the page content in a previously working state without reloading the whole application.

The *error.tsx* file exports a client component that the error boundary uses as the fallback interface. In other words, this component replaces the content when the code throws an error and activates the error boundary. As soon as it is active, it contains the error, ensuring that the layouts above the boundary remain active and maintain their internal state. The error component receives the error object and the reset function as parameters.

Let's add an error boundary to the *components/weather* route. Start by adding a new *ErrorProps* interface to type the component's properties into the *customs.d.ts* file:

```
interface ErrorProps {  
  error: Error;  
  reset: () => void;  
}
```

Next, create the *error.tsx* file next to *page.tsx* in the *app/components/weather* directory and add the code from Listing B-14.

```
"use client";  
  
export default function WeatherError(props: ErrorProps): JSX.Element {  
  return (  
    <section className="content error">  
      <span className="flag">app/components/weather/error(.tsx)</span>  
      <h2>Something went wrong!</h2>  
      <blockquote>{props.error?.toString()}</blockquote>  
      <button onClick={() => props.reset()}>Try again (re-render)</button>  
    </section>  
  );  
}
```

Listing B-14: The file app/components/weather/error.tsx adds the error boundary and the fallback UI.

Because we know that the error component needs to be a client component, we add the "use client" directive to the top of the file and then define and export the component. We use the *ErrorProps* interface we just created to type the component's properties. We then convert the error property to a string and display it to inform the user of the type of error that occurred. Finally, we render a button that calls the reset function that the component received through the properties object. The user can re-render the component into a previous working state by clicking the button.

Now, with the error boundary in place, we'll modify *component.tsx* to throw an error if the counter hits 4 or more. Open the file and add the code from Listing B-15 below the first *useEffect* hook.

```
useEffect(() => {
  if (count && count >= 4) {
    throw new Error("Count >= 4! ");
  }
}, [count]);
```

Listing B-15: The additional *useEffect* hook for *app/components/weather/component.tsx*

The additional *useEffect* hook we add to the component is straightforward; as soon as the *count* variable changes, we verify the error condition, and as soon as the variable's value is 4 or more, we throw an error with the message *Count >= 4!*, which the error boundary catches and gracefully handles by showing the fallback user interface that the *error.tsx* file exports.

To test this feature, open <http://localhost:3000/components/weather> in the browser and click the headline until you trigger the error. You should see the error component instead of the weather component, as in Figure B-5.

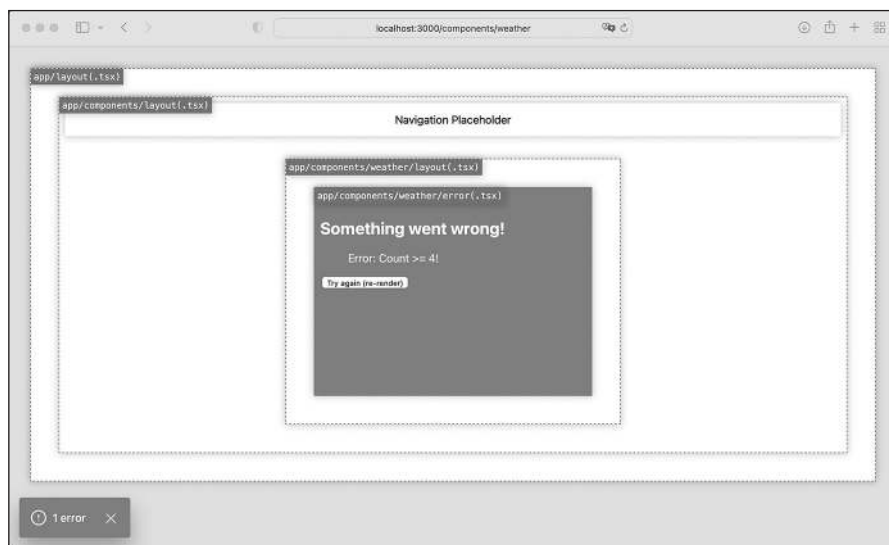


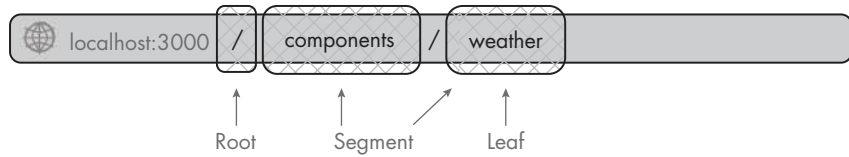
Figure B-5: The *components/weather* page in the error state

The layout markers show us that *error.tsx* has replaced *page.tsx*. We also see the string *Error: Count >=4!*, which we passed to the error constructor. As soon as we click the re-render button, *page.tsx* should replace *error.tsx*, and the screen will look like Figure B-4 previously.

Showing an Optional Loading Interface

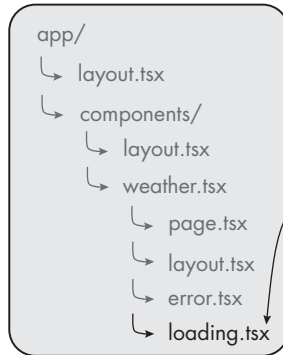
Now we'll create the *loading.tsx* file. With this feature in place, Next.js automatically wraps the page content with a React suspense component, creating a component hierarchy that looks similar to Figure B-6.

Browser URL:



Files:

Root: *app* directory



Simplified rendered structure

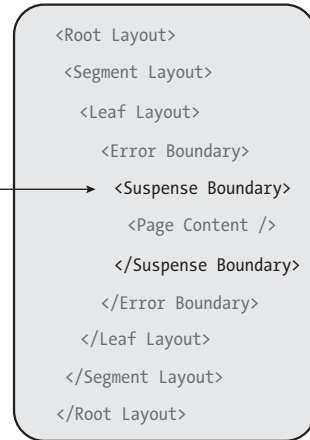


Figure B-6: The simplified layout component hierarchy with the loading interface

The *loading.tsx* file is a basic server component that returns the pre-rendered loading user interface. When we load a page or navigate between pages, Next.js will instantly display this component while loading the new segment's content. Once rendering is complete, the runtime will swap the loading state with the new content. In this way, we can easily display meaningful loading states, such as skeletons or custom animations.

Let's add a basic loading user interface to the weather component route by adding the code from Listing B-16 to the *loading.tsx* file.

```
export default function WeatherLoading(): JSX.Element {
  return (
    <section className="content">
      <span className="flag">app/components/weather/loading(.tsx)</span>
      <h1 className="loading"></h1>
    </section>
  );
}
```

Listing B-16: The file *app/components/weather/loading.tsx* adds a suspense boundary with the loading user interface.

We define and export the `WeatherLoading` component, which returns a `JSX.Element`. In the HTML, we add a headline element similar to the one in *page.tsx*, except this one adds the loading class we created in the *global.css* file to the headline and shows an animated placeholder.

When we open <http://localhost:3000/components/weather> in the browser, we should see a loading interface similar to Figure B-7.

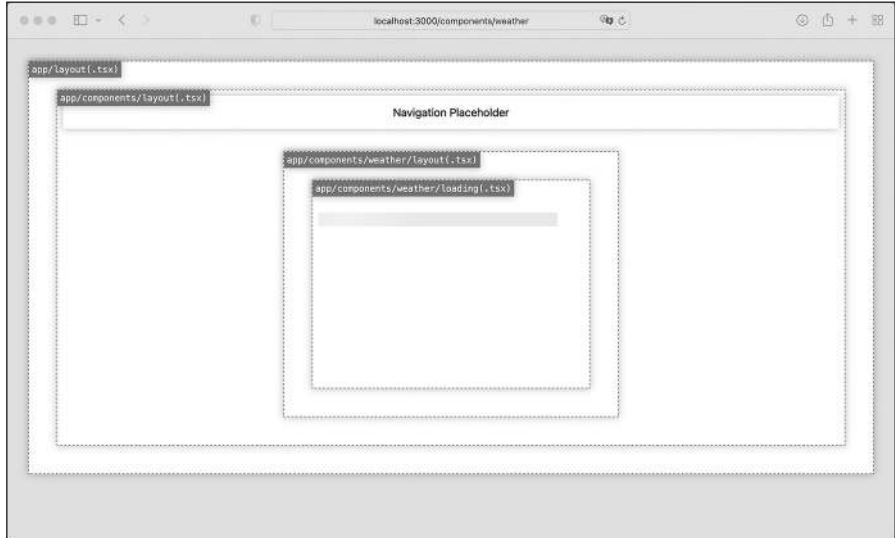


Figure B-7: The `components/weather` page while loading the page's content

If you don't see the animated placeholder, this means Next.js has already cached your segment's content.

Adding a Server Component That Fetches Remote Data

Now that you understand the folders and files in the *app* directory, let's add a server component that uses the `fetch` API to receive the list of users from the remote API <https://www.usemodernfullstack.dev/api/v1/users> and renders it to the browser. We wrote a version of this code in Chapter 5.

Create the folder *app/components/server-component* and add the special files *component.tsx*, *loading.tsx*, *error.tsx*, *layout.tsx*, and *page.tsx* to it. Then set up the component's functionality by adding the code from Listing B-17 to the *component.tsx* file.

```
export default async function ServerComponentUserList(): Promise<JSX.Element|Error> {
  const url = "https://www.usemodernfullstack.dev/api/v1/users";
  let data: responseItemType[] | [] = [];
  let names: responseItemType[] | [];
  try {
    const response = await fetch(url, { cache: "force-cache" });
    data = (await response.json()) as responseItemType[];
  }
```

```

    } catch (err) {
      throw new Error("Failed to fetch data");
    }
    names = data.map((item) => {
      return { id: item.id, name: item.name };
    });

    return (
      <ul>
        {names.map((item) => (
          <li id="{item.id}" key="{item.id}">
            {item.name}
          </li>
        ))}
      </ul>
    );
  }
}

```

Listing B-17: The app/components/server-component/component.tsx file

Here we create a default server component that uses the `fetch` API to await the API response. To be able to do so, we define it as an asynchronous function that returns a promise of a `JSX.Element` or an `Error`. Then we store the API endpoint in a constant and define the variables we'll need later on. We wrap the API call in a `try...catch` statement to activate the `Error Boundary` if the API request fails. We then transform the data in a manner similarly to the way we did in Chapter 5 and return a `JSX.Element` that displays a list of users.

Now we add the loading user interface that Next.js automatically displays while we await the API's response and the component's JSX response. Place the code from Listing B-18 into the *loading.tsx* file.

```

export default function ServerComponentLoading(): JSX.Element {
  return (
    <section className="content">
      <span className="flag">
        app/components/server-component/loading(.tsx)
      </span>
      <ul id="load">
        {[...new Array(10)].map((item, i) => (
          <li className="loading"></li>
        ))}
      </ul>
    </section>
  );
}

```

Listing B-18: The app/components/server-component/loading.tsx file

As before, the loading component is a server component that returns a `JSX.Element`. This time, the loading skeleton is a list with 10 items resembling the component's rendered HTML structure. You'll see that this gives the

user a good impression of the expected content and should improve the user's experience.

Next, we create the error boundary by adding the code from Listing B-19 to the *error.tsx* file.

```
"use client"; // Error components must be Client components

export default function ServerComponentError(props: ErrorProps): JSX.Element {
  return (
    <section className="content">
      <span className="flag">app/components/server-component/error(.tsx)</span>
      <h2>Something went wrong!</h2>
      <code>{props.error?.toString()}</code>
      <button onClick={() => props.reset()}>Try again (re-render)</button>
    </section>
  );
}
```

Listing B-19: The app/components/server-component/error.tsx file

Except for the flag outlining the file structure, the error boundary is similar to the one we used in the weather component.

Then we add the code from Listing B-20 to the *layout.tsx* file.

```
export default function ServerComponentLayout(props: LayoutProps): JSX.Element {
  return (
    <section>
      <span className="flag">app/components/server-component/layout(.tsx)</span>
      {props.children}
    </section>
  );
}
```

Listing B-20: The app/components/server-component/layout.tsx file

Again, the code is similar to the code we used for the weather component. We adjust only the flag outlining the component hierarchy.

Finally, with all the parts in place, we add the code from Listing B-21 to the *page.tsx* file to expose the page route.

```
import ServerComponentUserList from "../component";

export const metadata = {
  title: "Appendix C - Server Side Component (User API)",
  description: "The Example Code For A Server Side Component (User API)",
};

export default async function ServerComponentUserListPage(): JSX.Element {
  return (
    <section className="content">
      <span className="flag">app/components/server-component/page(.tsx)</span>
      { /* @ts-expect-error Async Server Component */ }
    </section>
  );
}
```

```

        <ServerComponentUserList />
    </section>
  );
}

```

Listing B-21: The app/components/server-component/page.tsx file

Completing the Application with the Navigation

With two pages in the application, we can now use the next/link component to replace the navigation placeholder in the nav element. This should create a fully functional application prototype that lets us navigate between the pages. Open the *app/components/layout.tsx* file and replace the code in the file with the code from Listing B-22.

```

import Link from "next/link";

export default function ComponentsLayout(props: LayoutProps): JSX.Element {
  return (
    <section>
      <span className="flag">app/components/layout(.tsx)</span>
      <nav>
        <Link href="/components/server-component">
          User API <br />
          (Server Component)
        </Link>{" "}
        |
        <Link href="/components/weather">
          Weather Component <br />
          (Client Component)
        </Link>
      </nav>
      <main>{props.children}</main>
    </section>
  );
}

```

Listing B-22: The updated app/components/layout.tsx file

We import the next/link component and then add two links to our navigation, one pointing to the user list server component we just created and the other pointing to the weather client component.

Let's visit the application's weather component page at *http://localhost:3000/components/weather*. You should see an application that looks similar to the screenshot in Figure B-8.

As soon as you navigate between the pages, you should see the loading user interface. With the outlines we've added to all the files, we easily keep track of which files Next.js uses to render the current page.

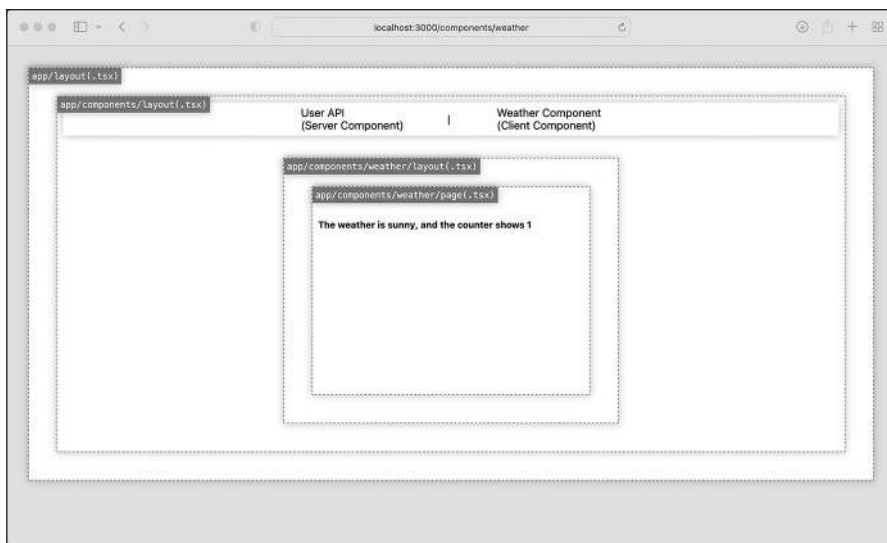


Figure B-8: The components/weather page with the functional navigation

Replacing API Routes with Route Handlers

If you look at the folder structure Next.js created for you, you should see that the `app` directory contains an `api` subfolder. You probably already guessed that we use this folder to define APIs. But unlike the API routes discussed in Chapter 5, which were regular functions, the `app` directory uses *route handlers*, which are functions that require a particular naming convention.

These route handlers belong in special files named *route.ts* that usually reside in a subfolder of the `app/api` folder. They are asynchronous functions that receive a Request object and an optional context object as parameters. We name each function after the HTTP method it should react to. For example, the code in Listing B-23 shows how to define route handlers that handle GET and POST requests for the same API.

```
import { NextRequest, NextResponse } from 'next/server';
export async function GET(request: NextRequest): Promise<NextResponse> {
  return NextResponse.json({});
}

export async function POST(request: NextRequest): Promise<NextResponse> {
  return NextResponse.json({});
}
```

Listing B-23: A skeleton structure of a route.ts file defining route handlers

To create the route handlers, we import the `NextRequest` and `NextResponse` objects from Next.js's server package. Next.js adds additional convenience

methods for cookie handling, redirects, and rewrites. You can read more about them in the official documentation at <https://nextjs.org>.

We then define two asynchronous functions, both of which receive a `NextRequest` and return a promise of a `NextResponse`. The function names correspond to the HTTP method they should respond to. Next.js supports using the GET, POST, PUT, PATCH, DELETE, HEAD, and OPTIONS methods as function names.

When defining page routes and route handlers, remember that these files take over all requests for a given segment. This means that a *route.ts* file cannot be in the same folder as a *page.tsx* file. Also, route handlers are the only way to define APIs if you use the *app* directory: you can't have an *api* folder in both the *pages* directory and *app* directory.

Next.js has statically and dynamically evaluated route handlers. Statically evaluated route handlers will be cached and reused for every request, whereas dynamically evaluated route handlers must request the data upon each request. By default, the runtime statically evaluates all GET route handlers that don't use a dynamic function or the `Response` object. As soon as we use a different HTTP method, the dynamic cookies or headers function, or the `Response` object, the route handler becomes dynamically evaluated. The same applies to APIs with dynamic segments, which receive the dynamic parameters through the context object.

Let's re-create the API route *api/v1/weather/[zipcode].ts* from Listing 5-1 as a route handler that we can use in the *app* directory. Add the code from Listing B-24 to a *route.ts* file in the folder structure *app/api/v1/weather/[zipcode]*.

```
import { NextResponse, NextRequest } from "next/server";

interface ReqContext {
  params: {
    zipcode: number;
  }
}

export async function GET( req: NextRequest, context: ReqContext ): Promise<NextResponse> {
  return NextResponse.json(
    {
      zipcode: context.params.zipcode,
      weather: "sunny",
      temp: 35,
    },
    { status: 200 }
  );
}
```

Listing B-24: The route handler in `app/api/v1/weather/[zipcode]/route.ts`

Notice that we've used the square brackets pattern on the folder structure to access the dynamic segment through the function's second parameter context object.

Within the file, we import the Next.js `Response` and `NextRequest` objects from the `server` package and then define the interfaces for the route handler. On the `RequestContext` interface, we add the `zipcode` property to `params`, representing the API's dynamic segment. Finally, we export the asynchronous `GET` function, the API route handler that reacts to all `GET` requests for this API endpoint. It receives the request object and the request context as parameters and uses the `NextResponse`'s `json` function to return the response data. We access the URL parameter `zipcode` through the context object's `params` object and then add it to the response data. We set additional response options through the `json` function's second parameter, explicitly setting the HTTP status code to `200`.

Now try querying the API with `curl`:

```
$ curl -i \  
  -X GET \  
  -H "Accept: application/json" \  
  http://localhost:3000/api/v1/weather/12345
```

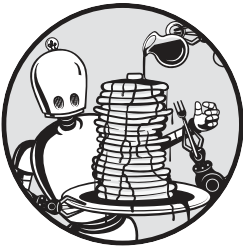
You should receive this JSON response:

```
HTTP/1.1 200 OK  
content-type: application/json  
--snip--  
{ "zipcode": "12345", "weather": "sunny", "temp": 35 }
```

This is the same response received in Chapter 5, where we accessed the API through the browser.



COMMON MATCHERS



In Jest, *matchers* let us check a specific condition, such as whether two values are equal or whether an HTML element exists in the current DOM. Jest comes with a set of built-in matchers. In addition, the *JEST-DOM* package from the testing library provides DOM-specific matchers.

Built-in Matchers

This section covers the most common built-in Jest matchers. You can find a complete list in the official JEST documentation at <https://jestjs.io/docs/expect>.

toBe This matcher is the simplest and by far the most common. It's a simple equality check to determine whether two values are identical. It behaves similarly to the strict equality (`===`) operator, as it considers type differences. Unlike the strict equality operator, however, it considers `+0` and `-0` to be different.

```
test('toBe', () => {
  expect(1+1).toBe(2);
})
```

toEqual We use `toEqual` to perform a deep-equality check between objects and arrays, comparing all of their properties or items. This matcher ignores undefined values and items. Furthermore, it does not check the object's types (for example, whether they are instances or children of the same class or parent object). If you require such a check, consider using the `toStrictEqual` matcher instead.

```
test('toEqual', () => {
  expect([undefined, 1]).toEqual([1]);
})
```

toStrictEqual The `toStrictEqual` matcher performs a structure and type comparison for objects and arrays; passing this test requires that the objects are of the same type. In addition, the matcher considers undefined values and undefined array items.

```
test('toStrictEqual', () => {
  expect([undefined, 1]).toStrictEqual([undefined, 1]);
})
```

toBeCloseTo For floating-point numbers, we use `toBeCloseTo` instead of `toBe`. This is because JavaScript's internal calculations of floating-point numbers are flawed, and this matcher considers those rounding errors.

```
test('toBeCloseTo', () => {
  expect(0.1+0.2).toBeCloseTo(0.3);
})
```

toBeGreaterThan/toBeGreaterThanOrEqual For numeric values, we use these matchers to verify that the result is greater than or equal to a value, similar to the `>` and `>=` operators.

```
test('toBeGreaterThan', () => {
  expect(1+1).toBeGreaterThan(1);
})
```

toBeLessThan/toBeLessThanOrEqual These are the opposite of the `GreaterThan...` matchers for numeric values, similar to the `<` and `<=` operators.

```
test('toBeLessThan', () => {
  expect(1+1).toBeLessThan(3);
})
```

toBeTruthy/toBeFalsy These matchers check if a value exists, regardless of its value. They consider the six JavaScript values 0, ' ', null, undefined, NaN, and false to be falsy and everything else to be truthy.

```
test('toBeTruthy', () => {
  expect(1+1).toBeTruthy();
})
```

toMatch This matcher accepts a string or a regular expression, then checks if a value contains the given string or if the regular expression returns the given result.

```
test('toMatch', () => {
  expect('apples and oranges').toMatch('apples');
})
```

toContain The toContain matcher is similar to toMatch, but it accepts either an array or a string and checks these for a given string value. When used on an array, the matcher verifies that the array contains the given string.

```
test('toMatch', () => {
  expect(['apples', 'oranges']).toContain('apples');
})
```

toThrow This matcher verifies that a function throws an error. The function being checked requires a wrapping function or the assertion will fail. We can pass it a string or a regular expression, similar to the toMatch function.

```
function functionThatThrows() {
  throw new Error();
}

test('toThrow', () => {
  expect(() => functionThatThrows()).toThrow();
})
```

The JEST-DOM Matchers

The *JEST-DOM* package provides matchers to work directly with the DOM, allowing us to easily write tests that run assertions on the DOM, such as checking for an element's presence, HTML contents, CSS classes, or attributes.

Say we want to check that our logo element has the class name center. Instead of manually checking for the presence of an element and then checking its class name attribute with `toMatch`, we can use the `toHaveClass` matcher, as shown in Listing C-1.

```


test('toHaveClass', () => {
  const element = getByTestId('image');
  expect(element).toHaveClass('center');
})
```

Listing C-1: The basic syntax for testing with the DOM

First we add the data attribute `testid` to our image element. Then, in the test, we get the element using this ID and store the reference in a constant. Finally, we use the `toHaveClass` matcher on the element's reference to see if the element's class names contain the class center.

Let's take a look at the most common DOM-related matchers.

getByTestId This matcher lets us directly access a DOM element and store a reference to it, which we then use with custom matchers to assert things about this element.

```


test('toHaveClass', () => {
  const element = getByTestId('image');
  --snip--
})
```

toBeInTheDocument This matcher verifies that an element was added to the document tree. This matcher works only on elements that are currently part of the DOM and ignores detached elements.

```


test('toHaveClass', () => {
  const element = getByTestId('image');
  expect(element).toBeInTheDocument();
})
```

toContainElement This matcher tests our assumptions about the element's child elements, letting us verify, for example, whether an element is a descendant of the first.

```
<div data-testid="parent">
  
</div>

test('toContainElement', () => {
  const parent = getByTestId('parent');
  const element = getByTestId('image');
  expect(parent).toContainElement(element);
})
```

toHaveAttribute This matcher lets us run assertions on the element's attributes, such as an image's alt attribute and the checked, disabled, or error state of form elements.

```


test('toHaveAttribute', () => {
  const element = getByTestId('image');
  expect(element).toHaveAttribute('alt', 'The Logo');
})
```

toHaveClass The `toHaveClass` matcher is a specific variant of the `toHaveAttribute` matcher. It lets us explicitly assert that an element has a particular class name, allowing us to write clean tests.

```


test('toHaveClass', () => {
  const element = getByTestId('image');
  expect(element).toHaveClass('center');
})
```

INDEX

SYMBOLS

- ` (backtick), 23
- \$ (dollar sign), 23
- => (fat arrow), 21
- ! (exclamation mark), 102
- . (period), 176
- + (plus operator), 34, 142
- ? (question mark), 45
- ... (spread operator), 27–28, 78
- [] (square brackets), 77, 102
- _ (underscore), 105

NUMBERS

- 200 status code, 107, 248–249
- 404 status code, 227
- 405 status code, 111
- 500 status code, 77, 101, 249

A

- absolute imports, 196
- abstract syntax tree (AST), 103–104
- access token, 159
 - using authorization grant to get, 171
 - using get protected resource, 172
- act, test cases, 133
- allowJs option, 259
- AMD format, 16
- anonymous functions, 16–17
- any type, 43
- APIs (application programming interfaces), 57
 - containers communicating through, 174
 - contracts, 34, 38, 94
 - GraphQL APIs, 101–113
 - microservices communicating through, 178
 - REST APIs, 93–101

routes

- creating, 90
- for GraphQL API, 110–111
- overview, 75–77
- replacing with route handlers, 285–287

Apollo sandbox, 111–113

Apollo server, 108

app directory, 72, 263–287

- exploring project structure, 269–287
 - adding content and route, 275–277
 - adding server component that fetches remote data, 281–284
 - catching errors, 277–279
 - completing application with navigation, 284–285
 - defining layout, 273–275
 - replacing API routes with route handlers, 285–287
 - showing optional loading interface, 279–281
 - updating CSS, 271–272
- rendering components
 - dynamic rendering, 268–269
 - fetching data, 266–267
 - static rendering, 267–268
- server components vs. client components
 - client components, 265
 - server components, 264–265

App function, 56

application glue, 120

application programming interfaces.

See APIs

apps

- serving from Docker container, 177
- using databases and object-relational mappers, 116

- arranging test cases, 132–133
- `array.map` function, 27
- arrays
 - dispersing, 27–28
 - identifying object types as, 102
 - looping through, 27
- array type, 41–42
- arrow functions, 20–22
 - exploring practical use cases, 22
 - lexical scope, 21–22
 - writing, 21
- assertion, test cases, 133–134
- AST (abstract syntax tree), 103–104
- asynchronous scripts
 - avoiding traditional callbacks, 24–25
 - simplifying, 26–27
 - using promises, 25–26
 - writing ES.Next module with, 29–30
- `async` keyword, 26–27
- audience claim, 164
- auditing *package.json* file, 10
- AuthElement component
 - adding to header, 241–243
 - overview, 238–240
- authentication
 - authorization vs., 158–159
 - REST APIs, 97–98
- authentication callback, 233–236
- auth guard, 248–249
- authorization, 157–172
 - accessing protected resource, 168–172
 - logging in to receive
 - authorization grant, 170–171
 - setting up client, 168–170
 - using access token to get
 - protected resource, 172
 - using authorization grant to
 - get access token, 171
- authentication vs., 158–159
- bearer tokens, 160–161
- code flow, 160–163
- creating JWT tokens, 163–168
 - header, 163
 - payload, 163–166
 - signature, 166–168
 - grant types, 159–160
 - role of OAuth, 159
- authorization code flow, 160–163
- authorization grant
 - logging in to receive, 170–171
 - using to get access token, 171
- authorization server, 159
- automated tests, 253–257
 - adding Jest to project, 254
 - setting up, 254–256
 - writing snapshot tests for header
 - element, 256–257
- `await` keyword, 26–27

B

- Babel.js, 15
- backend container
 - creating backend service, 187–189
 - seeding the database, 186–187
- backtick (```), 23
- `baseUrl` option, 259
- bearer tokens, 160–161
- `beforeAll` hook, 132
- `beforeEach` hook, 132
- black-box test, 144
- blocking time, 86
- block scope, 17
- Booleans, 40
- Boolean scalar type, 102
- built-in components
 - `next/head`, 80–81
 - `next/image`, 82–83
 - `next/link`, 81–82
- built-in hooks
 - handling side effects with
 - `useEffect`, 62–63
 - managing internal state with
 - `useState`, 62
 - sharing global data with
 - `useContext` and context providers, 63–64
- built-in matchers, 289–291
- built-in types
 - `any`, 43
 - `array`, 41–42
 - `object`, 42
 - primitive types, 40–41
 - `tuple`, 42–43

- union, 41
- void, 43–44
- built-in validators, 118
- button, generic, 235–238, 244–247

C

- @cacheControl directive, 103
- cacheControl.setCacheHint resolver
 - function, 103
- cached connection, 198
- callback hell, 25
- callbacks
 - array function running, 138
 - array.map function, 27
 - arrow functions simplifying, 22
 - avoiding traditional, 24–25
- callback URL, 162
- cases, test, 130
- catch all API route, 78
- catch method, 25–26
- claims, 163–166
 - private, 166
 - public, 165
 - registered, 164–165
- class components, 59–60
- client components, 265
- client credentials, 232
 - flow, 160
- client ID, 159
- clients, 159
- client secret, 159
- client-side rendering, 88–89
- cloning arrays and objects, 28
- code coverage, 130, 138–139
- code generator, 55
- collections, 117
- collisions, 161
- compilerOptions field, 37
- compilers, 36
- components, 57–61
 - Next.js built-in components, 80–83
 - next/head, 80–81
 - next/image, 82–83
 - next/link, 81–82
 - providing reusable behavior with
 - hooks, 61
 - styles for, 79–80
 - writing class components, 59–60

- concise body function, 21
- constant-like data, 20
- const keyword, 20
- constructor function, 59
- container class, 80
- containerization, 173–182. *See also*
 - Docker
- context providers, 63–64
- COPY keyword, 176
- create-next-app command, 70
- create-react-app command, 55
- Cross-Origin Resource Sharing (CORS), 75–76
- CRUD operations, 121–123, 199
- CSS styles, 78–80
 - adding to list item, 216–217
 - component styles, 79–80
 - global styles, 79
 - updating, 271–272
- cumulative layout shifts, 82
- curl command, 248, 251–252
- cURL tool, 99
- custom types, 44–45, 208
- Cypress, 253

D

- daemon service, 175
- database-connection middleware,
 - 120–121
- data mapping, 77
- data types, 20
- declarative programming, 54
- declaring variables, 17–20
 - constant-like data, 20
 - hoisted variables, 18–19
 - scope-abiding variables, 19
- default exports, 16–17
- default keyword, 16
- DefinitelyTyped repository, 46
- DELETE method, 98
- deleteOne function, 123
- deleting document, 123
- dependencies
 - installing, 8–9
 - overview, 6
 - removing, 11
 - replacing, 139–143
 - creating doubles folder, 141

- dependencies (*continued*)
 - replacing (*continued*)
 - creating module with
 - dependencies, 140–141
 - using fakes, 142
 - using mocks, 143
 - using stubs, 142
 - useEffect hook managing, 63
- details component, 227–228
- development dependencies
 - installing, 9–10
 - overview, 6
- development scripts, Next.js, 72
- directives, 208
- dispersing arrays and objects, 27–28
- Docker, 173–182, 185–193
 - building local environment with
 - backend container, 186–189
 - frontend container, 189–192
 - containerization architecture, 174
 - containers, 174–178
 - building Docker image, 176
 - interacting with, 178
 - locating exposed Docker port, 177–178
 - serving application from, 177
 - writing *Dockerfile*, 175–176
 - Docker Compose, 178–182
 - interacting with, 182
 - rerunning tests, 181–182
 - running containers, 180–181
 - writing *docker-compose.yml* file, 179–180
 - Food Finder application, 186
 - installing, 174
 - running automated tests in, 253–257
 - adding Jest to project, 254
 - setting up, 254–256
 - writing snapshot tests for
 - header element, 256–257
 - docker-compose.yml* file, 179–180
- document databases, 117
- document object model (DOM), 54, 57
- documents, 117
- dollar sign (\$), 23
- domain-specific language (DSL), 24
- doubles folder, 141

- dynamically typed languages, 34
- dynamic feedback, 38
- dynamic rendering, 268–269
- dynamic URLs, 77–78

E

- element constant, 57
- elements, 56
- encrypted tokens, 161
- endpoint, 95
- end-to-end query, 123–125
- end-to-end tests, 145, 151–153
- environment problems, 144
- errors, 133
 - catching, 277–279
 - with `const` keyword, 20
 - Internal Server Error, 77
 - non-hoisted variables, 19
 - promises and, 25–26
 - TypeScript, 36
 - using variables before declaring, 18
- escape character, 99
- `esModuleInterop` option, 259
- ES.Next modules, 15–17
 - importing modules, 17
 - using named and default exports, 16–17
 - writing with asynchronous code, 29–30
- exclamation mark (!), 102
- `exclude` option, 37
- executing script, using `npx`, 12
- `expect` function, 133–134
- expiration claim, 164
- `export` statement, 16
- exposed Docker port, 177–178
- ExpressJS Fundamentals course, 14
- Express.js server
 - building “Hello World,” 13–14
 - creating reactive user interface for, 64–67
 - extending with modern JavaScript, 29–31
 - extending with TypeScript, 46–51
 - adding type annotations to
 - index.ts* file, 49–50
 - adding type annotations to
 - routes.ts* file, 48–49

- creating *tsconfig.json* file, 47
- defining custom types, 47–48
- setting up, 46–47
- transpiling and running code, 50–51
- refactoring, 89–91
- extends option, 37
- external APIs, 93–94

F

- fakes, 142
- fat arrow (\Rightarrow), 21
- fetch API, 26–27, 266–267, 281–284
- Fibonacci sequence, 140–143
- fields, 117
- filter method, 22
- finally method, 25–26
- findOne function, 122
- Float scalar type, 102
- Food Finder application, 186
- forceConsistentCasingInFileNames
 - option, 260
- force flag, 10
- FROM keyword, 175
- frontend container
 - application service
 - adjusting for restarts, 191–192
 - creating, 189–190
 - global layout components, 222–226
 - header, 223–224
 - layout, 224–226
 - logo, 222–223
 - installing Next.js, 190–191
 - location details page, 227–230
 - start page, 216–222
 - list item component, 216–218
 - location list component, 218–219
 - user interface, 215–216
- fs module, 24–25
- functional tests, 144
- function components, 59
- functions
 - arrow, 20–22
 - exploring practical use cases, 22
 - lexical scope, 21–22
 - writing, 21

- avoiding traditional callbacks, 24–25
- type annotations declaring parameters of, 39–40
- function scope, 17–18

G

- gateway communications, 144–145
- generic button component, 235–238, 244–247
- getById matcher, 292
- GET method, 98–100
- getServerSideProps function, 84–85
- getToken function, 250
- GitHub OAuth app, 232
- global data, sharing with useContext
 - hook, 63–64
- global layout components, 222–226
 - header, 223–224
 - layout, 224–226
 - logo, 222–223
- global scope, 18, 44
- global styles, 79, 219–220
- Google Authenticator, 158
- Google scoring algorithm, 86
- gql tag, 210
- grant types, 159–160
- graph databases, 117
- GraphQL APIs, 75, 101–113, 207–214
 - adding API endpoint to Next.js, 212–214
 - adding to Next.js
 - adding data, 109
 - creating API route, 110–111
 - creating schema, 108–109
 - implementing resolvers, 109–110
 - using Apollo sandbox, 111–113
 - comparing REST to
 - over-fetching, 106–107
 - under-fetching, 107–108
 - connecting MongoDB to
 - adding services to GraphQL resolvers, 126–127
 - connecting to database, 125–126
 - merging typedefs into final schema, 209–210

- GraphQL APIs (*continued*)
 - resolvers, 103–106, 210–212
 - schemas, 101–103
 - custom types and directives, 208
 - mutation schema, 209
 - query schema, 209
 - securing mutations, 247–252
 - setting up, 208
- GraphQL queries, 209
- GraphQL schema, 24
- guards, 248

H

- hash-based message authentication
 - code (HMAC), 161
- Head elements, 80–81
- header
 - adding AuthElement component to, 241–243
 - global layout components, 223–224
 - JWT tokens, 163
 - writing snapshot tests for, 256–257
- hoisted variables, 18–19
- hooks, 62–64
 - handling side effects with `useEffect`, 62–63
 - managing internal state with `useState`, 62
 - providing reusable behavior with, 61
 - sharing global data with `useContext` and context providers, 63–64
- host system, 174–175
- hot-code reloading, 72
- HTML, 24
 - incremental static regeneration, 87
 - JSX elements and, 57
 - reactive user interface and, 54
 - static HTML exporting, 89
- HTTP methods, 98–99

I

- id helper program, 192
- ID scalar type, 102

- Image component, 82–83
- images, Docker, 176
- `` element, 82
- immutable data types, 20
- immutable elements, 57
- implicit flow, 160
- importing modules, 17
- import statement, 16–17
- include option, 37
- incremental option, 260
- incremental static regeneration (ISR), 87
- integration tests, 144–145
- interaction-based tests, 132
- interface keyword, 45
- interfaces
 - defining, 45
 - Mongoose model, 118
 - storing, 90
- inter-module communication, 144
- internal APIs, 93
- Internal Server Error, 77, 101
- internal state, managing with `useState` hook, 62
- Int scalar type, 102
- I/O operations, 24–25
- `isolatedModules` option, 260
- ISR (incremental static regeneration), 87
- issued at claim, 165
- issuer claim, 164

J

- JavaScript
 - arrow functions, 20–22
 - exploring practical use cases, 22
 - lexical scope, 21–22
 - writing, 21
 - asynchronous scripts
 - avoiding traditional callbacks, 24–25
 - simplifying, 26–27
 - using promises, 25–26
 - creating strings, 22–24
 - declaring variables, 17–20
 - dispersing arrays and objects, 27–28
 - ES.Next modules, 15–17, 29–30

- Express.js server
 - building “Hello World,” 13–14
 - extending, 29–31
 - looping through arrays, 27
 - Node.js, 3–14
 - creating projects, 8–12
 - installing, 4
 - package.json* file, 4–6
 - package-lock.json* file, 6–7
 - working with npm, 4
 - TypeScript, 33–51
 - benefits of, 34–36
 - built-in types, 40–44
 - custom types and interfaces, 44–46
 - extending Express.js server
 - with, 46–51
 - setting up, 36–38
 - type annotations, 38–40
 - JavaScript Syntax Extension (JSX)
 - example expression, 56–57
 - ReactDOM package, 57
 - JEST-DOM matchers, 292–293
 - Jest framework, 129–156
 - adding test cases to weather app
 - creating mocks to test
 - services, 148–151
 - evaluating user interface with
 - snapshot test, 153–156
 - performing end-to-end test of
 - REST API, 151–153
 - testing middleware with spies, 146–148
 - adding to project, 254
 - anatomy of test case
 - act, 133
 - arrange, 132–133
 - assertion, 133–134
 - creating example module to test, 131–132
 - matchers
 - built-in, 289–291
 - JEST-DOM, 292–293
 - replacing dependencies, 139–143
 - creating doubles folder, 141
 - creating module with
 - dependencies, 140–141
 - using fakes, 142
 - using mocks, 143
 - using stubs, 142
 - setting up, 130–131
 - test-driven development, 135–139
 - evaluating test coverage, 138–139
 - overview, 130
 - refactoring code, 136–138
 - types of tests
 - end-to-end tests, 145
 - functional tests, 144
 - integration tests, 144–145
 - snapshot tests, 145
 - unit testing, 130
 - jsonlint* package, 12
 - JSX. *See* JavaScript Syntax Extension
 - jsx option, 260
 - JWT (JSON Web Token)
 - defined, 160–161
 - header, 163
 - payload, 163–166
 - private claims, 166
 - public claims, 165
 - registered claims, 164–165
 - signature, 166–168
 - JWT claim, 165
- ## K
- key-value storage, 117
 - kill command, 178
- ## L
- layout
 - app* directory, 273–275
 - global layout components, 224–226
 - let keyword, 19
 - lexical scope, 21–22
 - lib option, 260
 - lifecycle methods, 59
 - Link component, 81–82
 - list item component, 216–218
 - loading user interface, 279–281
 - local environment
 - backend container, 186–189
 - creating backend service, 187–189
 - seeding the database, 186–187

- local environment (*continued*)
 - frontend container, 189–192
 - adjusting application service for restarts, 191–192
 - creating application service, 189–190
 - installing Next.js, 190–191
- location details page, 215
 - adding button to, 244–247
 - overview, 227–230
- location ID, 215
- location list component, 218–219
- location services
 - creating, 203–205
 - custom types for, 203
- logo, 222–223
- long-term support (LTS) version, 4
- looping through arrays, 27

M

- MAC (message authentication code), 161
- major version changes, 5
- matcher function, 134
- matchers
 - built-in, 289–291
 - JEST-DOM, 292–293
- Memcached, 117
- message authentication code (MAC), 161
- microservices, 178–182
 - interacting with Docker Compose, 182
 - rerunning tests, 181–182
 - running containers, 180–181
 - writing *docker-compose.yml* file, 179–180
- middleware, 120–121, 195–206
 - configuring Next.js to use absolute imports, 196
 - connecting Mongoose, 196–199
 - fixing TypeScript warning, 198–199
 - writing database connection, 197–198
 - creating Mongoose model
 - creating location model, 201–202
 - creating schema, 199–200
- model services, 202–206
 - creating location services, 203–205
 - testing, 206
 - testing with spies, 146–148
- minor version changes, 5
- mobile-first design pattern, 222
- mocks, 143, 148–151
- module option, 260
- moduleResolution option, 260
- module scope, 18, 44
- MongoDB, 101, 115–128
 - connecting GraphQL API to database, 125–126
 - adding services to GraphQL resolvers, 126–127
 - creating end-to-end query, 123–125
 - defining Mongoose model
 - database-connection
 - middleware, 120–121
 - interfaces, 118
 - model, 119–120
 - schema, 118–119
 - how apps use databases and object-relational mappers, 116
 - querying database
 - creating document, 121–122
 - deleting document, 123
 - reading document, 122
 - updating document, 122–123
 - relational and non-relational databases, 116–117
 - setting up Mongoose and, 117
- MongoDB Query Language (MQL), 117
- Mongoose
 - connecting middleware, 196–199
 - fixing the TypeScript warning, 198–199
 - writing database connection, 197–198
 - creating model
 - creating location model, 201–202
 - creating schema, 199–200
 - defining model
 - database-connection
 - middleware, 120–121
 - interfaces, 118

- model, 119–120
- schema, 118–119
- setting up, 117
- MQL (MongoDB Query Language), 117
- multifactor authentication, 158
- mutations, 101, 211–212
 - defining schema, 209
 - input type object for, 102–103
 - securing GraphQL, 247–252
- MySQL, 101

N

- named exports, 16–17
- name field, 5
- name flag, 177
- navigation, 284–285
- Neo4j, 117
- nested page routes, 73–75
- networking protocols, 8
- next-auth, 231–235
 - adding client credentials, 232
 - creating authentication callback, 233–236
 - creating GitHub OAuth app, 232
 - installing, 233
 - sharing session across pages and components, 235
- next export command, 89
- next/head component, 80–81
- next/image component, 82–83
- Next.js, 13, 69–91
 - adding API endpoint to, 212–214
 - adding GraphQL API to, 108–113
 - adding data, 109
 - creating API route, 110–111
 - creating schema, 108–109
 - implementing resolvers, 109–110
 - using Apollo sandbox, 111–113
 - app* directory, 263–287
 - exploring project structure, 269–287
 - rendering components, 266–269
 - server components vs. client components, 264–265
 - built-in components
 - next/head, 80–81

- next/image, 82–83
- next/link, 81–82
- configuring to use absolute imports, 196
- installing in container, 190–191
- pre-rendering and publishing, 83–89
 - client-side rendering, 88–89
 - incremental static regeneration, 87
 - server-side rendering, 84–85
 - static HTML exporting, 89
 - static site generation, 86–87
- refactoring React and Express.js applications, 89–91
- routing applications, 72–78
 - API routes, 75–77
 - dynamic URLs, 77–78
 - nested page routes, 73–75
 - simple page routes, 73
- setting up, 70–72
 - development scripts, 72
 - project structure, 71–72
- styling applications, 78–80
 - component styles, 79–80
 - global styles, 79
- wish list page, 243–244
- next/link component, 81–82
- Node.js, 3–14
 - creating projects, 8–12
 - auditing *package.json* file, 10
 - cleaning up *node_modules* folder, 11
 - executing script only once using npx, 12
 - initializing new module or project, 8
 - installing dependencies, 8–9, 11–12
 - installing development dependencies, 9–10
 - removing dependencies, 11
 - updating all packages, 11
 - Express.js-based Node.js server, 13–14
 - installing, 4
 - package.json* file, 4–6
 - dependencies, 6

- Node.js (*continued*)
 - package.json* file (*continued*)
 - development dependencies, 6
 - required fields, 5
 - package-lock.json* file, 6–7
 - TypeScript installation in, 36–37
 - working with npm, 4
- node_modules* folder
 - cleaning up, 11
 - package.json* file vs., 4–5
- node package execute (npx) tool, 12
- noEmit option, 260
- non-hoisted variables, 19–20
- non-nullable fields, 102
- non-primitive data types, 20
- non-relational databases, 116–117
- NoSQL databases, 117
- not before claim, 165
- npm, 4
 - npm audit command, 10
 - npm init command, 8
 - npm install command, 7, 11–12
 - npm prune command, 11
 - npm run build command, 72
 - npm test command, 131
 - npm uninstall command, 11
 - npm update command, 11
 - npx command, 70
 - npx next build command, 72
 - npx tool, 12
 - null types, 40–41
 - numbers, as primitive types, 40
- O**
 - OAuth, 157–172
 - accessing protected resource
 - logging in to receive
 - authorization grant, 170–171
 - setting up client, 168–170
 - using access token to get
 - protected resource, 172
 - using authorization grant to
 - get access token, 171
 - adding button to location detail
 - component, 244–247
 - adding with next-auth, 231–235
 - adding client credentials, 232
 - creating authentication
 - callback, 233–236
 - creating GitHub OAuth
 - app, 232
 - installing next-auth, 233
 - sharing session across
 - pages and components, 235
 - AuthElement component
 - adding to header, 241–243
 - overview, 238–240
 - authentication vs., 158–159
 - authorization code flow, 161–163
 - bearer tokens, 160–161
 - creating JWT tokens
 - header, 163
 - payload, 163–166
 - signature, 166–168
 - generic button component, 235–238
 - grant types, 159–160
 - role of OAuth, 159
 - securing GraphQL mutations, 247–252
 - wish list Next.js page, 243–244
- object data modeling, 116
- object-relational mappers, 116
- objects, 27–28
- object type, 42
- one-time password (OTP), 158
- online playground, 37, 55
- online registry, npm, 4
- OpenAPI format, 95
- over-fetching, 106–107
- over-typing, 38–39
- P**
 - package.json* file, 4–6
 - auditing, 10
 - dependencies, 6
 - development dependencies, 6
 - editing, 29
 - required fields, 5
 - package-lock.json* file, 6–7
 - packages
 - npm online registry, 4
 - updating, 11

- page routes
 - adding, 275–277
 - creating, 90–91
 - nested, 73–75
 - simple, 73
- pages* folder, 71–72
- parameters of functions, 39–40
- PATCH method, 98
- patch version changes, 5
- \$PATH environment variable, 12
- payload, JWT tokens, 163–166
 - private claims, 166
 - public claims, 165
 - registered claims, 164–165
- period (.), 176
- persisting the data, 116
- Playwright, 253
- plus operator (+), 34, 142
- POST method, 98
- prefixes, 79–80
- pre-rendering, 83–89
 - client-side rendering, 88–89
 - incremental static regeneration, 87
 - server-side rendering, 84–85
 - static HTML exporting, 89
 - static site generation, 86–87
- primitive types, 20, 40–41
- private APIs, 93
- private claims, 166
- profile pages, 77
- promise chain, 26
- Promise object, 25
- props argument, 57–58, 84, 86
- protected resource
 - logging in to receive authorization
 - grant, 170–171
 - setting up client, 168–170
 - using access token to get protected
 - resource, 172
 - using authorization grant to get
 - access token, 171
- providers, 231
- public claims, 165
- public* folder, 71
- publish-all flag, 177
- push method, 20
- PUT method, 98

Q

- queries, 101
- querying database, 121–123
- query schema, 209
- question mark (?), 45

R

- ReactDOM package, 57
- reactive user interface, 54
- React, 53–67
 - creating reactive user interface for
 - Express.js server, 64–67
 - JavaScript Syntax Extension,
 - 56–57
 - organizing code into components,
 - 57–61
 - providing reusable behavior
 - with hooks, 61
 - writing class components,
 - 59–60
 - refactoring, 89–91
 - role of, 53–55
 - setting up, 55–56
 - working with built-in hooks
 - handling side effects with
 - useEffect, 62–63
 - managing internal state with
 - useState, 62
 - sharing global data with
 - useContext and context
 - providers, 63–64
- reading
 - data, 99–100
 - document, 122
 - files, 25
- Redis, 117
- refactoring code, 136–138
- refresh token, 160
- registered claims, 164–165
- relational databases, 116–117
- remote data, fetching, 281–284
- rendering components
 - dynamic rendering, 268–269
 - fetching data, 266–267
 - static rendering, 267–268
- replacing dependencies, 139–143
 - creating doubles folder, 141

- replacing dependencies (*continued*)
 - creating module with
 - dependencies, 140–141
 - using fakes, 142
 - using mocks, 143
 - using stubs, 142
- replay attack, 165
- report, test-coverage, 138–139
- require statement, 16
- resolveJsonModule option, 260
- resolvers, 210–212
 - implementing, 109–110
 - overview, 103–106
- resource owner, 159–160
- resource providers, 159
- REST APIs, 75, 93–101, 212
 - comparing GraphQL to
 - over-fetching, 106–107
 - under-fetching, 107–108
 - creating end-to-end query, 123–125
 - HTTP methods, 98–99
 - overview, 94–95
 - performing end-to-end test of,
 - 151–153
 - specification, 95–97
 - state and authentication, 97–98
 - URLs, 95
 - working with
 - reading data, 99–100
 - updating data, 100–101
- restarts, 191–192
- RESTful APIs, 159
- return value, type annotations
 - declaring, 39
- reusable behavior, 61
- root entry point, 95
- root privileges, 192
- route handlers, 285–287
- routing applications, 72–78
 - API routes, 75–77
 - dynamic URLs, 77–78
 - nested page routes, 73–75
 - simple page routes, 73

S

- save-dev flag, 36–37, 46
- scaffolding process, 55
- scalar types, 102

- Schema Definition Language (SDL), 101
- schemas
 - GraphQL APIs, 101–103, 108–109
 - custom types and directives, 208
 - merging typedefs into final
 - schema, 209–210
 - mutation schema, 209
 - query schema, 209
- Mongoose, 118–119, 199–200
- scope, variable, 17
- scope-abiding variables, 19
- scope property, 22
- screenshots, 145
- script, executing once using npx, 12
- SDL (Schema Definition Language), 101
- secrets, 233
- securing GraphQL mutations, 247–252
- seeding the database, 124, 186–187
- semantic versioning, 5–6
- SEO metadata, 80, 86
- server components, 264–265
- server-side rendering (SSR), 84–85
- services, 121
- session information, 97–98
- sessions, sharing across pages and
 - components, 235
- SHA-256 hash algorithm, 161
- side effects, 62–63
- SIGKILL command, 182
- signature, JWT tokens, 166–168
- signed tokens, 161
- SIGTERM command, 182
- single-factor authentication, 158
- skipLibCheck option, 260
- snapshot tests, 145
 - evaluating user interface with,
 - 153–156
 - writing for header, 256–257
- specification, 95–97
- spies, 146–148
- spread operator (...), 27–28, 78
- SQL (Structured Query Language),
 - 116–117
- square brackets ([]), 77, 102
- SSG (static site generation), 86–87, 215
- SSR (server-side rendering), 84–85
- SSR (static site rendering), 216, 244

- start page, 216–222
 - list item component, 216–218
 - location list component, 218–219
- state-based tests, 132
- stateless, REST APIs, 97–98
- statically typed languages, 36
- static exports
 - API routes and, 76
 - HTML, 89
- static HTML file, 66
- static rendering, 267–268
- static site generation (SSG), 86–87, 215
- static site rendering (SSR), 216, 244
- steps, test, 130
- sticky header, 222–223
- strings
 - benefits of TypeScript, 34
 - creating, 22–24
 - as primitive types, 40
 - template literals for, 22–24
- String scalar type, 102
- Structured Query Language (SQL), 116–117
- stubs, 142
- styles* folder, 71–72
- styles object, 80
- styling applications, 78–80
 - component styles, 79–80
 - global styles, 79
- subject claim, 164
- suites, test, 130
- sum function, 131–132, 135
- super function, 59–60
- Swagger, 95–97

T

- tag flag, 176
- tagged template literal, 22–24
- target option, 260
- template literals, 22–24
- test-coverage report, 138–139
- test doubles, 139
- test-driven development (TDD), 135–139
 - evaluating test coverage, 138–139
 - overview, 130
 - refactoring code, 136–138
- testing, 129–156
 - adding test cases to weather app
 - creating mocks to test the services, 148–151
 - evaluating user interface with snapshot test, 153–156
 - performing end-to-end test of REST API, 151–153
 - testing middleware with spies, 146–148
 - anatomy of test case
 - act, 133
 - arrange, 132–133
 - assertion, 133–134
 - creating example module to test, 131–132
 - model services, 206
 - replacing dependencies, 139–143
 - creating doubles folder, 141
 - creating module with dependencies, 140–141
 - using fakes, 142
 - using mocks, 143
 - using stubs, 142
 - rerunning, 181–182
 - running automated tests in Docker, 253–257
 - adding Jest to project, 254
 - setting up, 254–256
 - writing snapshot tests for header element, 256–257
 - setting up, 130–131
 - test-driven development, 135–139
 - evaluating test coverage, 138–139
 - overview, 130
 - refactoring code, 136–138
 - types of
 - end-to-end tests, 145
 - functional tests, 144
 - integration tests, 144–145
 - snapshot tests, 145
 - unit testing, 130
- testing-library/dom* assert package, 134
- testing-library/react* assert package, 134
- test runner, 130
- testWatch command, 254–256
- then method, 25–26

- third-party APIs, 93–94
- this keyword, 21–22, 59
- time to first paint, 86
- toBeCloseTo matcher, 290
- toBeGreaterThan/toBeGreater
ThanOrEqual matcher, 290
- toBeInTheDocument matcher, 292
- toBeLessThan/toBeLessThanOrEqual
matcher, 290–291
- toBe matcher, 289–290
- toBeTruthy/toBeFalsy matcher, 291
- toContainElement matcher, 293
- toContain matcher, 291
- toEqual matcher, 290
- toHaveAttribute matcher, 293
- toHaveClass matcher, 293
- toMatch matcher, 291
- toStrictEqual matcher, 290
- toThrow matcher, 291
- transpilers, 36
- TSC. *See* TypeScript Compiler
- tsconfig.json* file, 37–38
- tuple type, 42–43
- type annotations, 38–40
 - declaring parameters of functions,
39–40
 - declaring return value, 39
 - declaring variables, 39
- type declaration files, 45–46
- typedefs, 101, 209–210
- type keyword, 44–45
- TypeScript, 16, 33–51
 - benefits of, 34–36
 - built-in types
 - any, 43
 - array, 41–42
 - object, 42
 - primitive types, 40–41
 - tuple, 42–43
 - union, 41
 - void, 43–44
 - custom types and interfaces
 - defining custom types, 44–45
 - defining interfaces, 45
 - using type declaration files,
45–46
 - extending Express.js server with,
46–51
 - setting up
 - dynamic feedback, 38
 - installation in Node.js, 36–37
 - tsconfig.json* file, 37–38
 - type annotations, 38–40
 - declaring parameters of
functions, 39–40
 - declaring return value, 39
 - declaring variables, 39
 - using JSX with, 57
- TypeScript Compiler (TSC), 36
 - fixing warning, 198–199
 - options, 259–261
- @types scope, 46

U

- UMD format, 16
- undefined type, 40–41
- under-fetching, 107–108
- underscore (*_*), 105
- union type, 41
- unit testing, 130
- untagged template literal, 22–23
- updateOne function, 122–123
- useContext hook, 63–64
- useEffect hook, 61–63, 89, 244
- user ID, 215, 244
- user interfaces
 - evaluating with snapshot tests,
153–156
 - frontend container, 215–216
 - showing optional loading user
interface, 279–281
- user property, 192
- useSession hook, 240
- useState hook, 61–62, 89

V

- v (version) flag, 4
- variables, 17–20
 - constant-like data, 20
 - hoisted variables, 18–19
 - scope-abiding variables, 19
 - type annotations declaring, 39
- var keyword, 18–19
- version field, 5
- versioning APIs, 95
- viewport, 81

- virtual DOM, 54
- Visual Studio Code, 38
- void type, 43–44
- volume flag, 177
- volumes, 177
- vulnerabilities, 10

W

- W3Schools tutorials, 14, 67
- weather app
 - creating mocks to test the services, 148–151
 - evaluating user interface with snapshot test, 153–156

- performing end-to-end test of REST API, 151–153
- testing middleware with spies, 146–148
- wish list Next.js page, 243–244
- WORKDIR keyword, 175

Y

- YAML, 188
- yarn, 4